Effective Resource Utilization for Multiprocessor Join Execution*

Marguerite C. Murphy‡
Doron Rotem
Computer Science Research Dept.
Lawrence Berkeley Laboratory
Berkeley, CA 94720

Abstract

Conventional approaches to execution of database queries on general purpose multiprocessors attempt to maximize system throughput using inter-query parallelism with a fixed number of processors. Standard uniprocessor optimization techniques are used to minimize execution time of individual queries. Our approach is to increase performance by utilizing intra-query parallelism as well as minimizing overall resource requirements. Specifically, *processor* and *i/o bandwidth* requirements are minimized by coordinating the order in which data pages are read into memory and page joins assigned to available processors. We present a scheduling strategy based on join indices and prove lower and upper bounds on its resource requirements. We then describe a heuristic for estimating the number of processors required to complete join execution in minimal time. Our simulation results indicate that these techniques are effective with respect to processor utilization and buffer requirements.

1. Introduction

Multiprocessors have recently entered the marketplace as a cost effective high performance alternative to high-end mainframe uniprocessors. Multiprocessor architectures exploit current microprocessor technology by integrating a variable number of processors into a single system with all processors sharing a single main memory and input/output subsystem. High performance is achieved primarily via parallelism and to a lesser extent by sharing of information in main memory. With coarse grained parallelism (i.e. at the process level) existing applications can be executed concurrently without modification by simply assigning each process to a separate processor (assuming the use of system-call compatible operating systems). For example, any of the major UNIX based relational DBMS (Ingres, Oracle, Informix, Unify) can be executed with minimal modifications on a SEQUENT multiprocessor [SEQU88]. Inter-query parallelism is achieved by running multiple (independent) queries in parallel on multiple processors, using standard uniprocessor algorithms and optimization techniques.

Although impressive increases in system throughput can be achieved with coarse grained parallelism, it does not take advantage of the potential concurrency within the individual processes themselves. There are two types of parallelism that might be realized: parallel execution of CPU operations on multiple processors and overlap of CPU and I/O operations. Since the degree of CPU parallelism is potentially unbounded, system performance rapidly becomes limited by the i/o bandwidth--the rate at which data can be transferred to and from the stable storage devices into main memory for computation. In this paper we continue our investigation [MURP89] into new algorithms for parallel execution of relational join operations within the limits imposed by the i/o subsystem bandwidth.

Our approach is based on a decomposition of relational join processing into a collection of page reads and page joins (with dependencies introduced by the requirement that data pages be read into memory before participating in any page joins). We assume that data on secondary storage is organized into fixed size pages with an indexing scheme that allows construction of a "page connectivity graph"-- a bipartite graph with one node corresponding to each page of each relation and one edge connecting each pair of pages which contain at least one matching join attribute value [MERR81]. Page connectivity graphs can be easily constructed from join indices [VALD87], Bc trees [GOYA88] or intermediate results computed during index join processing [BLAS76]. The granularity of processor scheduling is the individual page join and the granularity of i/o scheduling the individual page read.

Given a page connectivity graph describing the page joins to be performed, we first determine how many processors to allocate to the computation, then schedule the order in which data pages are read into memory from disk and finally schedule each join for execution on a processor. Initially we assume that sufficient memory is available to buffer pages until joining pages and processors are

---

* Issued as tech report LBL-26601. This work is supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy under Contract DE-AC03-76SF00098. Authors' electronic mail addresses: rotem@csam.lbl.gov, murphy@lbl-csam.arpa

‡ Also with Computer Science Department, San Francisco State University.

Proceedings of the Fifteenth International Conference on Very Large Data Bases

Amsterdam, 1989

available, implying that a page never needs to be read into memory more than once. In [MURP89] we presented a family of practical read scheduling algorithms to use with FIFO processor scheduling.

In [MURP89] we also present lower bounds on the join execution time as well as bounds on the number of processors required to complete processing in minimum time. These bounds assumed page join processing times were constant and equal to the page read time. In this paper we relax this assumption and present analogous bounds for arbitrary constant page join times as well as a more detailed stochastic performance model which includes both constant and exponentially distributed page join times. In addition, we present a heuristic for estimating the actual number of processors to allocate to a particular join graph and introduce a modified scheduling algorithm to be used in an environment with a bounded buffer pool size.

To summarize, the major contributions of this paper are: (1) a heuristic for estimating the optimal number of processors for a particular join graph, (2) a modified scheduling algorithm to be used in environments with a bounded buffer pool size, (3) bounds on execution time and number of processors assuming constant page join times, (4) extensive simulation results corroborating algorithms, bounds and heuristics.

This paper is organized as follows. The next section summarizes our join processing strategy and the following section our improved bounds on resource requirements. Section four presents a performance model based on queueing theory and the simulation model we implemented. Section five summarizes the results of a series of experiments designed to evaluate the effectiveness of our strategy. In the final section we present our conclusions and recommendations for practical implementation.

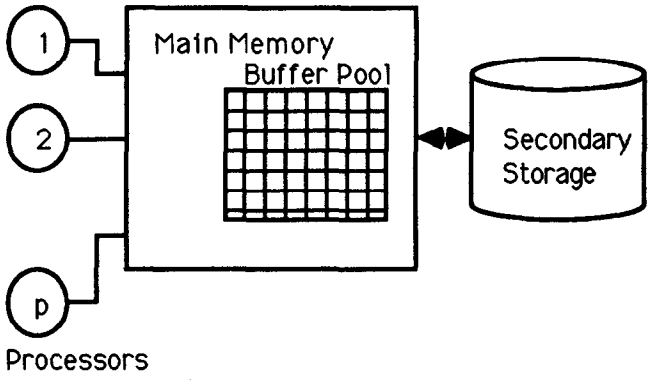
2. Join Processing Strategy

In this section we present a condensed review of material presented in [MURP89]. We do this in order to more clearly explain the extensions introduced in this paper as well as to make this paper self-contained. After describing our basic scheduling algorithm, we present a heuristic extension for use in environments with a bounded number of buffers available for join processing.

2.1 Multiprocessor Architecture

We assume a multiprocessor architecture with a flexible number of processors, one main memory and a single shared i/o subsystem. Each processor executes instructions independently of the other processors and can be individually scheduled. Data is stored and transferred in fixed sized units, referred to as blocks on secondary storage, buffers in main memory or simply pages of data. Data is transferred between the i/o subsystem and memory over a channel which has a concurrency of one (that is, at most one i/o operation can be in progress at any time). Once a data page is resident in a buffer, it can be accessed by any number of processors simultaneously. Note that we do not require simultaneous access to individual addressable units of main memory.

The following figure summarizes the basic system architecture, which is typical of that found in existing commercial multiprocessors, i.e. [SEQU88b].



2.2 Preliminaries

The following standard definitions and notations from graph theory are used in the remainder of this paper. A bipartite graph B(V,E) consists of a vertex set $V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$ and edge set $D \subseteq V_1 \times V_2$, i.e. edges are pairs of the form (x,y) where $x \in V_1$ and $y \in V_2$. A complete bipartite graph is a bipartite graph with the maximum number of edges, i.e., $|E| = |V1| * |V2|$.

For a vertex in V, we denote by d(v) its degree, which is equal to the number of edges incident on it. A connected component C of B is a subgraph of B such that a path exists between every pair of vertices of C and no path exists between a vertex in B-C and a vertex in C. We can find all the connected components of a graph B in $O(|V| + |E|)$ time using a depth-first search algorithm [Aho74].

2.3 Read & Processor Scheduling

We summarize here the scheduling algorithm presented in [MURP89]. The algorithm takes a bipartite graph B(V,E) as its input and produces a schedule S for p processors from it. For expository reasons we divide the process of producing a schedule into four stages. A schedule consists of a read schedule and a join schedule. The first three stages are directed towards producing an efficient read schedule. In the final step we perform the join schedule.

In stage 1, we decompose the graph into its connected components. In stage 2, for each component C_i we derive independently an ordering of its page reads. In stage 3, we compute for each component C_i with m edges and n vertices the function $f(C_i) = m / p - n$. We then concatenate the read schedules of all the components in decreasing order of $f(C_i)$.

At this point we have a read schedule for the whole graph. It now remains to schedule the page joins at each step. This is done in stage 4 in a simple manner. We maintain a queue of unprocessed joins as follows. Initially the queue is empty. Let us assume that vertex v is scheduled to be read at time step i. We remove from the queue the first p edges and schedule their corresponding page joins on processors 1,2,...,p respectively. If the queue contains less than p edges then some (or all) of the processors remain idle at this step. We then insert into the

end of the queue all the edges incident on v which have their other endpoint memory resident. By our definitions there are A(v) such edges.

As stages 1, 3 and 4 are relatively simple we only give here a more formal description of stage 2, which is presented as Algorithm 1.

Algorithm 1:

Input: Component C with vertex set V and edge set E.

Output: A read schedule S for C. We build S by concatenating a new vertex to it each time the loop is processed. the variable A(v) keeps track of the actual work for each node. The degree of node v is denoted by d(v).

Step 1: (Initialize) For all $v \in V$, set $A(v) := 0$ and $S := \emptyset$;

Step 2: (First Vertex) Choose a vertex v_x such that $d(v_x) = \max \{ d(v) \text{ where } v \in V\}$, ties may be broken arbitrarily.

Step 3: (Add chosen vertex to S) Set *last* := v_x, append *last* to S.

Step 4: (Update actual): for all $v \in V\text{-}S$, where v is adjacent to *last* set $A(v) := A(v) + 1$;

While $V\text{-}S \neq \emptyset$ do
BEGIN

Step 5: (Choose all vertices with maximum A(v))Let MAX be the set of all vertices not in S with maximum A(v).

MAX := { $v \in V\text{-}S \mid A(v) = \max \{A(v) \text{ and } v \in V\text{-}S\}$}

Step 6: (Maximize potential) Set *last* :=w where $w \in$ MAX with the value of d(w) as large as possible (break ties arbitrarily). Append *last* to S.

Step 7: (Update actual) For all $v \in V\text{-}S$ update actual work as in Step 4.

END

2.4 Restrictions on the Number of Buffers

In case the number of available buffers is restricted to some constant K, we may be forced to replace pages in memory before all their associated joins are completed. Such pages will have to be read again (at least once) to complete the execution of all their joins. We are interested in identifying join graphs which may force such a replacement. For such graphs it is not possible to complete the join execution using only |V| input operations.

In the next theorem we show a connection between the structure of the join graph, the constant K, and page replacements. The following definitions are used in the theorem. A vertex subgraph of a bipartite graph $B(V_1,V_2,E)$, is itself a bipartite graph $B'(V_1',V_2',E')$ where V_i' is a subset of V_i (i=1,2) and E' consists of all edges in E with both endpoints in V_i'. For a graph X, we denote by min(X) the value of the smallest degree of a vertex in X.

Theorem 1: Given a join graph B with |V| vertices, a join execution for B requires at least |V| +1 read operations under a restriction of K buffers, if B contains a vertex subgraph B' such that $\min(B') \geq K$.

Proof: Let us assume that B contains a subgraph B' with $\min(B') \geq K$ and $|V| = r$. We observe that $r \geq 2K$. We will derive a contradiction by assuming that it is possible to complete the join of B with no page replacements.

Let us label the nodes of B' as $v_1,v_2,\ldots,v_r$ such that v_i is the ith node of B' read by the schedule. When v_r, the last vertex of B' is read in, it must join with at least K other nodes from B' by our assumption on min(B'). Since there are at most K-1 available buffers, at least one of these neighbors is not currently present in memory. This means that at least one additional node of B' must be read in contradicting the fact that v_r is the last node of B' read by the schedule. □

The above theorem is of theoretical interest only since identifying such a subgraph is an NP-complete problem [GARE79]. In the next section we show how our heuristic algorithm can be modified to operate efficiently under buffer restrictions.

2.5 Modifications to the Heuristic for Buffer Restrictions

In the presence of memory restrictions, Algorithm 1 has to be modified so that nodes can be replaced in memory and then read again. In this section we describe informally how our scheduling can be dynamically adapted to an environment with only K buffers. In this case, some changes must be made to the read schedule when it cannot simply read a new node in because all K buffers contain nodes which must still participate in more joins. At this point we have to make a choice between two alternatives:

(a) In the next step no node will be read in and only joins among memory resident nodes will be performed;

(b) Choose a node in memory (the "victim") and replace it by an input node. The "victim" node will be reread at some later point in order to complete its joins.

Our strategy, as before, is to attempt at maximizing the amount of work in the system by "greedy" decisions. For each node v we keep track of its actual and potential work when it is read in using the counters A(v) and P(v) respectively. We subtract one from each of these counters with each join performed which involves node v. A node for which P(v) becomes zero can be replaced by a new node without any rereads.

As in Algorithm 1, we read in the next node for which A(v) is maximum as long as we have free buffers. Let us assume that the next input node according to our read schedule is x, and there are no free buffers, i.e., the current set of nodes in memory is M where |M| = K and no member v of M has P(v) = 0. We compute the value of the total maximum actual work over all nodes in M and compare it with the total actual work which can be achieved in the system by replacing a node in M with x. If the former value is larger, we simply proceed with alternative (a). Otherwise we choose a node y in memory for which the total actual

work in the system (computed after replacement of y with x) is the largest possible. We then read node x into the buffer currently occupied by node y.

The following adjustments must be made:

* All joins involving node y must be removed from the work queue.

* The value of A(v) must be decremented for all neighbors of node y.

* The node y is placed in the queue of unread nodes with its current value of A(y) and P(y) and will be scheduled for reading according to our usual criteria.

* As before, after node x is read in, we need to update the values of A(v) for all nodes in the system which are neighbors of x (increase by 1) and add all joins involving x and a memory resident node into the work queue.

3. Bounds Assuming Constant Page Join Processing Time

In this section we assume that the join graph B(V,E) is a bipartite connected graph where $V = V_1 \cup V_2$. The bounds we derive here are based on knowledge of some simple parameters of the graph such as the cardinalities of the edge and vertex set or the size of the largest degree in the graph. Of course it is possible to derive tighter bounds if we have more information about the graph. Bounds are important since they give us some ideal measures against which we can compare our algorithms. We consider in this section bounds on the number of processors required to complete the join in optimal time. In the next section we deal with bounds on the time required to complete the join with a given fixed number of processors.

We denote by $T_{opt}(B)$ the optimal time to complete a join represented by the graph B, i.e. the minimum number of time steps with an unbounded number of processors where the join time is a constant C.

Lemma 1:

$$T_{opt}(B) = |V| + C.$$

Proof: Let S be an optimal time schedule for the graph B. At each time step we can read exactly one vertex of the graph. Let us denote by v_n the last vertex read by the schedule S. The processing of all edges incident on v_n can be completed only after v_n has been read. Since the graph is connected, there will be at least one edge incident on v_n and therefore at least C additional time units are needed after all the vertices have been read in order to complete the execution.

In the following theorems we compute an upper and a lower bound on $P_{opt}(B)$, the number of processors required to complete the join in optimal time $T_{opt}(B)$. We derive a bound which assumes that all we know about the graph is |V| and |E|.

Lemma 2: The maximum number of page joins a schedule can complete during the first i + C time units (with unbounded number of processors) is

$$\begin{cases} \frac{i^2}{4} \text{ for i even} \\ \frac{(i-1)(i+1)}{4} \text{ for i odd} \end{cases}$$

Proof: Let us denote by $B_i(S)$ the subgraph of B read by a schedule S during its first i steps. This subgraph consists of the set of i vertices read by the schedule and all edges of B with both endpoints in this set. Let V^i_1 and V^i_2 be the vertices of $B_i(S)$ which belong to V_1 and V_2 respectively. At step i+1, the schedule S can perform all joins such that their corresponding edges are in $B_i(S)$. It is easy to see that for any schedule S, the number of edges in $B_i(S)$ is maximized when this graph is a complete bipartite graph with $|V^i_1| = |V^i_2| = i/2$ when i is even and $|V^i_1| = (i-1)/2$ and $|V^i_2| = (i+1)/2$ for i odd. The expression in the statement of the lemma represents the number of edges in the graph corresponding to each of these cases. □

Theorem 2: The number of processors required to complete the join in optimal time on the graph B(V,E) satisfies

$$P_{opt}(B) \geq C \times MAX\left\{\frac{|E|}{t}; \frac{|V| - \sqrt{|V|^2 - 4|E|}}{2}\right\}$$

where $t = |V| + C - 2$.

Proof: We first prove that the number of processors needed is larger than the first term in the curly brackets. Let us assume that schedule S completes the join in optimal execution time with $P_{opt}(B)$ processors. We observe that there are t time units in which processors must complete all |E| joins because no join can take place during the first two time units. Since a single processor can complete at most $\left\lfloor \frac{t}{C} \right\rfloor$ joins during this period we have

$$P_{opt}(B) \geq \frac{|E|\,C}{t}$$

We now prove that the number of processors needed is also larger than the second term in the curly brackets. We denote by c(i) the number of page joins performed by S during its first i+C time units. Then in order for S to complete the join in optimal time, it has to perform the additional |E|-c(i) joins during the remaining time which is |V| - i. Since each processor can complete at most $\left\lfloor \frac{|V| - i}{C} \right\rfloor$ joins during this period, we have the inequality

$$P_{opt}(B) \geq C \times \left\lceil \frac{|E| - c(i)}{|V| - i} \right\rceil$$

By Lemma 2, for $1 \leq i \leq |V|$

$$c(i) \leq \frac{i^2}{4}$$

From which we conclude that for $1 \leq i \leq |V|$

$$P_{opt}(B) \geq C \times \left(\frac{|E| - \frac{i^2}{4}}{|V| - i} \right)$$

In order to make the bound as tight as possible we will find the value of i for which the right hand side achieves its

maximum. We use elementary calculus to find that the right hand side achieves its maximum when the value of i is the closest integer to

$$|V| - \sqrt{|V|^2 - 4|E|}$$

By substituting this value of i into the right hand side we obtain

$$P_{opt}(B) \geq \frac{|V| - \sqrt{|V|^2 - 4|E|}}{2}$$

as claimed.□

In the next theorem we exploit more information about the graph to derive an upper bound on $P_{opt}(B)$. We assume that $|E|$, $|V_1|$, $|V_2|$ are given. Without loss of generality let $|V_1| \leq |V_2|$.

Theorem 3: The optimal number of processors satisfies

$$P_{opt}(B) \leq \left\lceil \frac{|E|}{|V_2|} \times M \right\rceil$$

where $M = \lceil C \rceil$ if $C \geq 1$ and $M = \frac{1}{\lfloor \frac{1}{C} \rfloor}$ if $C < 1$.

Proof: We exhibit a simple schedule S' which completes the join in optimal time using no more than the above number of processors. The schedule S' is characterized by the following rules: Let us call a step in which a vertex of V_1 is read a type I step and all other steps are called type II steps.

3.1 The vertices of V_2 are sorted in non-increasing order of their degrees and relabelled $v_1, v_2, \ldots, v_n$ such that v_1 is the vertex with the maximum degree and v_n has the minimum degree.
The schedule performs its reads in n+1 rounds each consisting of a type II step followed by zero or more type I steps in the following way:

3.2 In the first time step of round i ($i<n+1$) v_i is read, this is followed by reading in all the vertices of V_1 connected to it which have not yet been read. Round n+1 consists of the final join.

3.3 All page joins are scheduled to take place as early as possible, i.e. as soon as the two endpoints of an edge have been read in and there is a free processor to perform the join.

The number of new potential joins introduced at the end of round i is at most equal to $d(v_i)$, the degree of v_i. For simplicity, from now on we will assume that page joins are performed only on the first step of each round (from round 2 onwards), i.e. on the first step of round i+1 we will attempt to perform all remaining page joins involving v_i. (this is possible as all endpoints of edges involved have been read). The proof of the theorem has two parts, in Part I we prove that it is sufficient to complete at each round $A = \lceil \frac{E}{n} \rceil$ joins in order to obtain optimal execution time. We then show in Part II that A x M processors are sufficient to achieve this goal.

Part I:

Intuitively, this result holds since there are $|E|$ joins which must be performed over n rounds and therefore A is roughly the average number of joins per round. More precisely, at the beginning of round i+1 (for $i<n$) there are at most $d(v_i)$ new joins to be performed. In case only A are performed during this round, there are up to $d(v_i)-A$ potential joins which may have to be deferred to some future round.

Note that the value of A is the ceiling of the average degree of vertices in the set V_2, so that there must be a first index j such that $d(v_j) \leq A$. At the beginning of round j, the total number of page joins deferred from all previous rounds is at most

$$\sum_{i=1}^{i=j-1} (d(v_i)-A)$$

On the other hand, by the decreasing order of degrees, there will be a total of at least

$$\sum_{i=j}^{n} (A-d(v_i))$$

available processors to complete these deferred joins during rounds $j+1, j+2, \ldots, n+1$. Since as we noted before A is equal or larger than the average degree in the set V_2 we have

$$A \geq \frac{1}{n} \sum_{i=1}^{i=n} d(v_i)$$

from which it follows that

$$\sum_{i=j}^{n} (A - d(v_i)) \geq \sum_{i=1}^{i=j-1} (d(v_i) - A)$$

and the number of available processors is sufficient to complete all the page joins.

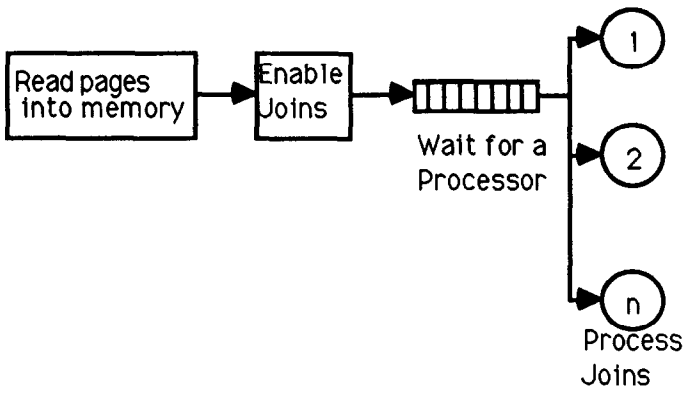
Part II:

Given a join time of C we will now show how the processors can be scheduled during each round. If $C \leq 1$ each processor can complete $\lfloor \frac{1}{C} \rfloor$ joins in one time unit. Therefore A x M processors are sufficient to complete A joins in each round. In the case $C > 1$, we can use a group of A processors at the beginning of each round. Since a group of processors is utilized for at most C time units we can reuse it every $\lceil C \rceil$ steps. In this way we will be using at most A x M processors at any given step.□

4. Discrete Event Simulation

The dynamic behavior of our algorithm can be modeled by a simple multi-server queueing model. At each step of the read schedule a data page is read into memory. After the read is complete, one or more joins to other memory resident data pages may be enabled. As soon as a join is enabled, it may be assigned to a processor for execution. If all processors are busy, the enabled joins are queued until

they can be processed. The following diagram summarizes this behavior:



In order to evaluate our algorithms and heuristics, we implemented a stochastic simulation model. Input parameters are summarized in the following table:

Input	Description
NumNodes	Total Number of Nodes in Graph
Relation 1	Number of nodes in first Relation (Opt)
Alpha	Fraction of Edges present in graph
Mean	Mean CPU Processing Time
Distribution	Exponential/Constant Processing Time
Seed	Random Number Seed
Runs	Number of Replications
Lookahead	Depth of Potential Cost Evaluation
NumProcs	Number of Processors (Optional)

The first step of our simulation is to generate a *random graph* by partitioning the nodes into two subsets. If the number of nodes in the first subset is not specified, a random partition is performed. Next the number of edges is determined as the product of alpha and the maximum number of edges possible. We randomly select this number of edges (without replacement) from all possible edges and construct a bipartite graph. This graph represents a page connectivity graph. The page connectivity graph is then used as input to our scheduling algorithm to produce a page read schedule giving the order in which pages (nodes) should be read into memory. This schedule is augmented by a list of joins (edges) ordered by the times at which they become enabled.

This join list is then input into a discrete event simulation [LAWK82] of a multiprocessor with n processors. Each processor has a join processing time (service time) described by a random variable. The random variable has either a constant or exponential distribution with a fixed mean. Throughout the simulation, random numbers are generated using techniques described in [PARK88]. If the number of processors is not specified, the simulation will estimate the number of processors required to complete processing in optimal time (Popt) and use this number during simulation of join execution.

The following table lists the output parameters of our simulation:

Output	Description
Popt	Estimated Optimal Number of Procs
Execution Time	Simulated total execution time
Utilization	Simulated Processor Utilization
NumBufs	Number of Buffers required

4.1 Optimal Number of Processors

The optimal number of processors is estimated by assuming that the page join times are constant and equal to the page read times. In [MURP89] we presented lower bounds on the execution time in terms of the total number of nodes (|V|), the number of edges (|E|) and the number of processors (p):

$$\text{Execution time} \geq \max\left\{|V| + 1,\ 2p + 1 + \left\lceil \frac{|E| - p^2}{p} \right\rceil \right\}$$

In initial experiments we observed that the simulated execution time was almost always equal to the lower bound. Intuitively this should be the case when the processor utilization is sufficiently low that joins rarely need to wait to be processed. We then estimate the optimal number of processors as the value of p for which

$$|V| + 1 = 2p + 1 + \left\lceil \frac{|E| - p^2}{p} \right\rceil$$

In the following experiments we evaluate the use of this estimate when the page join times are constant, but not necessarily equal to the page read time. Intuitively, if the page join times are close to the page read times, the estimated optimal number of processors should be close to the true optimal value.

5. Simulation Experiments and Results

In this section we present the results of an extensive series of simulation experiments designed to verify our algorithms, bounds and heuristics. The charts displayed in this paper are representitive of the results we obtained.

5.1 Comparison with Random Schedule

In this experiment we compared the execution time produced by the read schedule of Algorithm 1 with a random read schedule for three typical graphs. As we can see in Charts 1 and 2, Algorithm 1 is consistently better (by roughly 20%) when the number of processors is less than the optimal. As expected, increasing the number of processors above Popt has little effect on the execution time since no additional improvement is possible.

5.2 Execution Time and Popt

This experiment measured the execution time produced by Algorithm 1 for a wide range of typical join graphs. The parameters of these graphs are given in the two tables below. The trends in Charts 3 and 4 are consistent with our predictions, i.e. the execution time decreases with the number of processors and increases with the number of edges. The charts level off at the optimal execution time

slightly before we use Popt processors, i.e. no further improvements in execution time occur after that point.

Graph	Symbol
Split=45-55%, Alpha = .25	◆
Split=45-55%,Alpha = .60	◇
Split=25-75%,Alpha = .25	■
Split=25-75%, Alpha = .60	□

Number of Nodes	Split	Alpha	Popt
75	45-55%	0.25	7
	25-75%	0.25	5
	45-55%	0.6	15
	25-75%	0.6	11
25	45-55%	0.25	3
	28-67%	0.25	3
	45-55%	0.6	6
	28-67%	0.6	5

5.3 Buffers and Popt

Our simulation model assumes that a buffer is freed whenever all joins associated with the node have been completed. In this experiment we measure the maximum number of buffers used by Algorithm 1 under this assumption. We observe a few interesting trends in these Charts (5 and 6):

(1) The relative sizes of the relations and densities of the graphs is significant, i.e. graphs which are more evenly split and/or denser require more buffers.

(2) Adding more processors helps to reduce the number of required buffers as joins have to wait less time for execution. If the number of processors is less than Popt, the number of buffers required increases sharply.

5.4 Join Time and Popt

In the following series of experiments we vary the page join times between 0.1 and 2.0 (the units are fraction of the time required for a page read). In this experiment we measured the execution time (Charts 7 and 9) and processor utilization (Charts 8 and 10) for the optimal number of processors (Popt), the optimal number of processors plus a constant one (Popt +1) and the optimal number of processors less one (Popt -1). We observe that the two metrics are highly correlated with the number of processors. The execution time is constant (and bounded by the optimal time) until the processors "saturate" (i.e. utilization approaches one), at which point the execution time increases roughly linearly with the page join time.

5.5 Exponential vs Constant Join Times

These experiments were designed to examine the effects on processor utilization and total execution time of stochastic variations in the page join times (Charts 11,12,13,14). We compared the performance of two typical join graphs assuming exponentially distributed join times with the performance using constant page join times having the same means. We did not find significant differences in execution time or processor utilization, indicating that our bounds based on an assumption of constant page joins times are reasonable approximations to the situation of exponentially distributed page join times.

6. Conclusions and Future Work

In this paper we studied the problem of optimizing join execution on multiprocessors from both the theoretical and practical point of view. We derived lower bounds on the execution time and optimal number of processors based on the structure of the join graph. We then devised a heuristic scheduling algorithm which produces an order of reading in the pages on the relations and scheduling the joins on the processors. We identified several parameters which might significantly affect the resource utilization requirements of a join plan. These include: size of the graph, density of edges, relative sizes of relations, distribution of join time and number of processors. We conducted a large number of experiments with our heuristic in order to examine the effect of all these variables on the resource requirements and their inter-dependencies.

Our results indicate that the heuristic performs well under a wide range of conditions and the resource utilization achieved by it matches quite closely the theoretical lower bounds.

Future work in this area includes examining more join strategies and also algorithms for additional relational operators. In the near future we plan to integrate buffer restrictions and heuristic strategies into our simulation models. In addition, we plan to implement our algorithms and obtain performance measurements on a commercial multiprocessor such as Sequent.

We believe that in the future query optimizers for databases which run on multiprocessors will have to be enhanced in order to take full advantage of the parallelism offered by such a system. Such optimizers will need to incorporate cost functions based on accurate predictions of resource requirements and execution time in order to correctly evaluate the costs associated with different join plans. The research presented here is a first step in this direction.

7. References

[Aho74] Aho, A. V., J. E. Hopcroft and J. D. Ullman. (1974). The Design and Analysis of Computer Algorithms. Reading, Mass, Addison-Wesley.

[BLAS76] Blasgen, M. W. and K. P. Eswaran. (1976). On the Evaluation of Queries in a Relational Data Base System. IBM Technical Report #RJ1745 (#25553) Computer Science (April 8, 1976).

[GARE79] Garey, M. and D. Johnson. (1979). Computers and Intractability. San Francisco, W.H. Freeman and Company.

[GOYA88] Goyal, P., H. F. Li, E. Regener and F. Sadri. (1988)."Scheduling of Page Fetches in Join Operations Using Bc-Trees." Proceedings of 4th International Conference on Data Engineering,Los Angeles, CA,IEEE.

[LAWK82] Law, A. M. and W. D. Kelton. (1982). Simulation Modeling and Analysis. New York, McGraw Hill.

[MERR81] Merrett, T., Y. Kambayashi and H. Yasuura. (1981)."Scheduling of Page-fetches in Join Operations." Proceedings 7th International Conference on Very Large Data Bases,Cannes, France.

[MURP89] Murphy, M. C. and D. Rotem. (1989)."Processor Scheduling for Multiprocessor Joins." Fifth International Conference on Data Engineering,Los Angeles, CA,IEEE.

[PARK88] Park, S. K. and K. W. Miller. (1988). "Random Number Generators: Good ones are Hard to Find." CACM. 31(10).

[SEQU88] Sequent Computer and Codd & Date Associates. (1988). Combining the Benefits of Relational Database Technology and Parallel Computing. Technical Seminar, San Francisco, CA, September 28, 1988.

[SEQU88b] Sequent Computer Systems. (1988). Systems Overview. Product Description, Sequent Computer, Beaverton, ORE.

[VALD87] Valduriez, P. (June 1987). "Join Indices." ACM Transactions on Database Systems. 12(2): 218-246.

Chart 1:

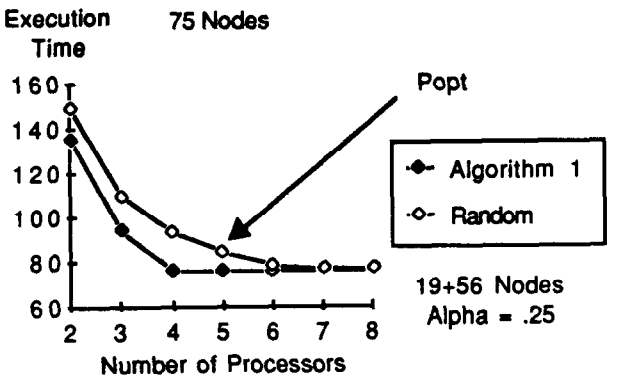


Chart 2:

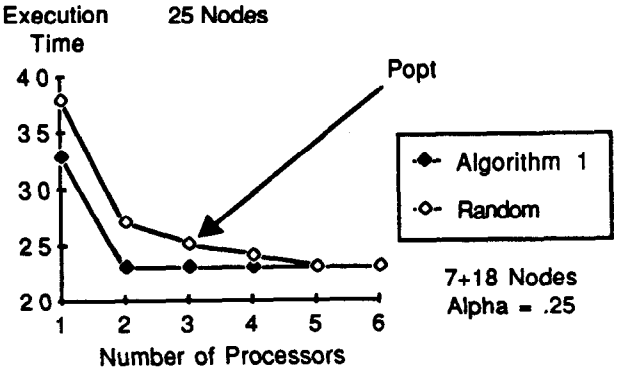


Chart 3:

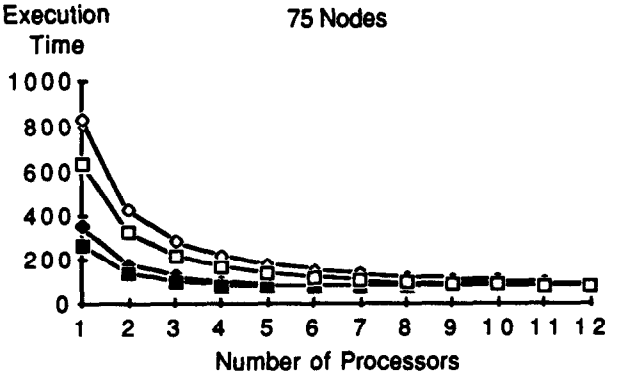


Chart 4:

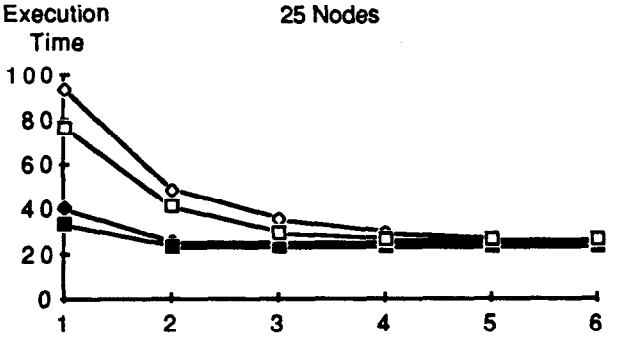


Chart 5:

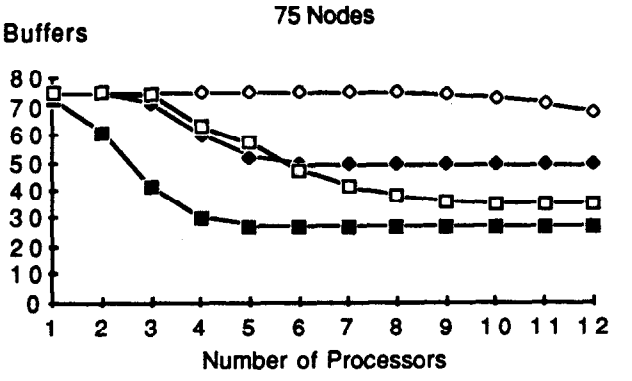


Chart 6:

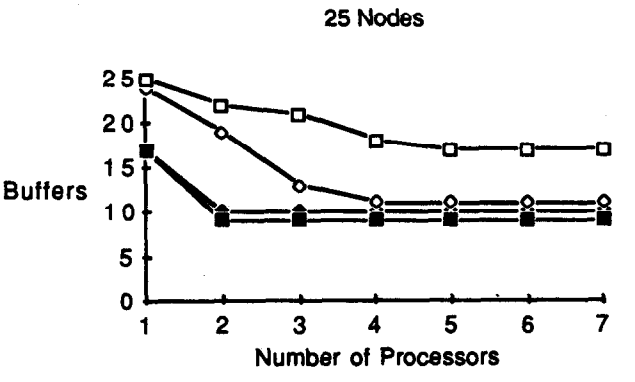


Chart 7:

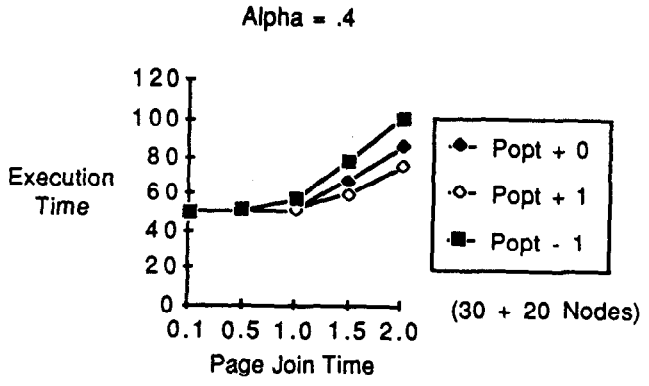


Chart 8:

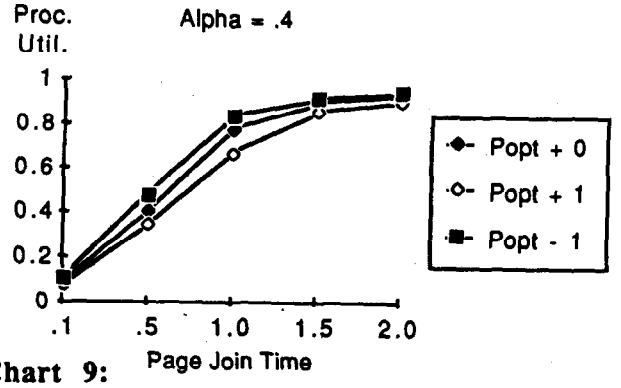


Chart 9:

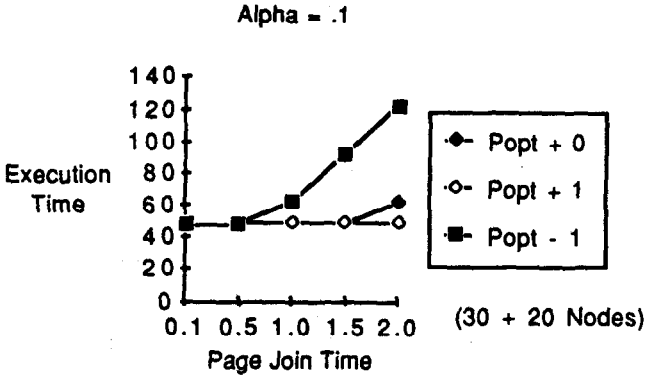


Chart 10:

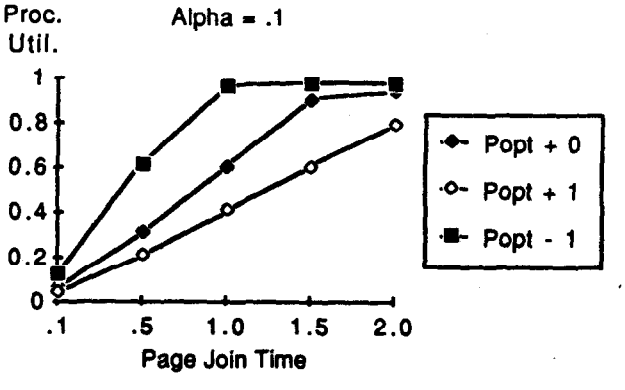


Chart 11:

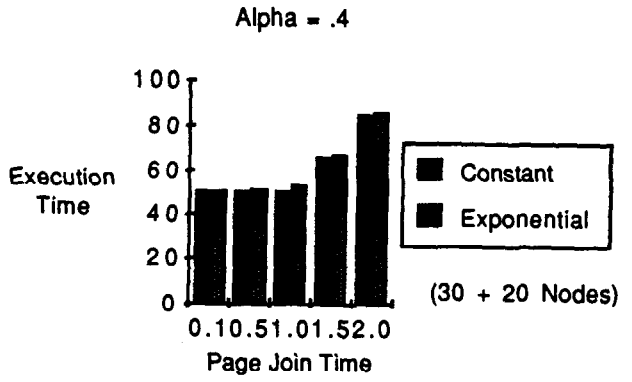


Chart 12:

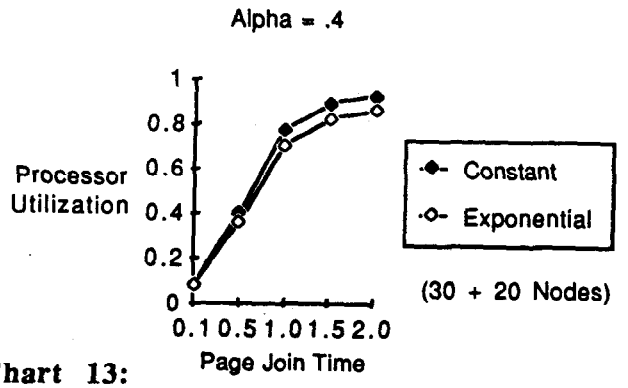


Chart 13:

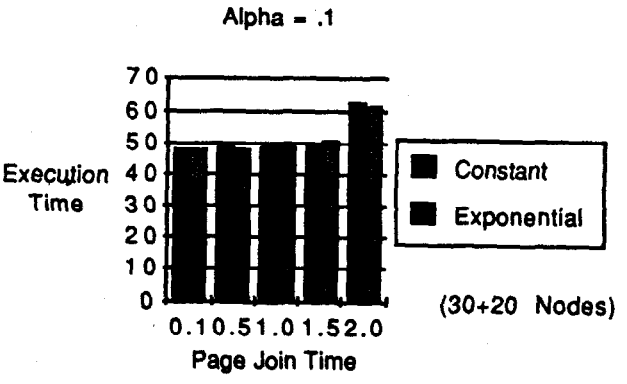


Chart 14: