Nested Loops Revisited

David J. DeWitt* Jeffrey F. Naughton*† Joseph Burger*

Abstract

The research community has considered hash-based parallel join algorithms the algorithms of choice for almost a decade. However, almost none of the commercial parallel database systems use hashing-based join algorithms, using instead nested-loops with index or sort-merge. While the research literature abounds with comparisons between the various hash-based and sort-merge join algorithms, to our knowledge there is no published comparison between the parallel hash-based algorithms and a parallel nested loops algorithm with index. In this paper we present a comparison of four variants of parallel index nested loops algorithms with the parallel hybrid hash algorithm. The conclusions of our experiments both with an analytic model and with an implementation in the Gamma parallel database system are that (1) overall, parallel hybrid hash is the method of choice, but (2) there are cases where nested-loops with index wins big enough that systems could profit from implementing both algorithms. Furthermore, our experiments show that among the nested loop algorithms, one of them, *subset nested loops with sorting*, clearly dominates.

1 Introduction

The research community has long considered hash-based joins to be the method of choice for performing joins in multiprocessor database systems. One line of reasoning behind this goes roughly like this: it has long been known that in uniprocessor systems, sort-merge beats nested loops with index almost always [BE77]; also, hybrid-hash beats sort-merge just about everywhere in both the uniprocessor and multiprocessor case [DKO+84, SD89]; so, transitively, one can expect hybrid hash to out-perform nested-loops with index. Furthermore, the hybrid hash and sort merge algorithms can be used to execute any join in a query, not just joins in which at least one of the two relations being joined is a base relation.

However, in the multiprocessor case, an interesting asymmetry arises in the nested-loops with index algorithm: only one of the operand relations in a two-relation join needs to be partitioned over the network. This suggests that parallel nested loops with index may be able to exploit this asymmetry in a parallel environment to provide higher performance than parallel hybrid hash, which partitions both relations in the case that neither relation has been declustered on its joining attribute.

It was our goal in this research to explore the performance of the parallel nested-loops with index join algorithm, to determine:

1. Are there significant differences between variants of parallel index nested loops?

2. Are there cases where parallel nested loops with index provides better performance than parallel hybrid hash?

3. If there are such cases, how significant is the performance difference in these cases?

4. When parallel hybrid-hash does perform better than parallel nested loops with index, how significant is the difference?

The answers to these questions help to determine which of these join algorithms needs to be included in a parallel database system. This is of more than academic interest; we are aware of at least two commercial parallel DBMSs that employ parallel nested loops with index.

We used a two-pronged approach in order to answer these questions. First, we built a simple analytic model predicting the performance of the parallel hybrid hash join and four versions of parallel index nested loops. This analytic model was useful in quickly exploring the

*Department of Computer Sciences, University of Wisconsin-Madison. This research was supported by donations from DEC, IBM (through an IBM Research Initiation Grant), NCR, and Tandem.

†Work supported in part by NSF grant IRI-9157357.

relative performance of the algorithms for a wide range of hardware and join operand parameters. However, in our opinion an analytic model alone can never provide the detailed insight and confidence that is engendered by an actual implementation. Accordingly, we also implemented the algorithms on the Gamma parallel database system [DGS+90].

The results from our analytic model and from our implementation both demonstrate that parallel nested loops with index can perform much better than parallel hybrid hash if (1) one of the relations is small, and (2) either the other relation has a clustered index on the join attribute, or the other relation has an index on the join attribute and fits entirely in memory. In all other cases, hybrid hash is significantly better than the parallel nested loops algorithms. Furthermore, the results show that one version of the parallel index nested loops algorithms, *subset nested loops with sorting*, clearly dominates the others.

In related work, DeWitt and Gerber investigated the performance of four parallel hashing join algorithms [DG85], while Schneider and DeWitt compared several parallel hashing algorithms with parallel sort-merge [SD89]. Kitsuregawa [KTMo83] proposed an algorithm based on hashing for redistribution followed by sort-merge, intended to take advantage of special-purpose sorting hardware. None of these papers compared the hashing join algorithms with nested loop with index algorithms. Valduriez and Gardarin [VG84] compared parallel join and semijoin algorithms based on hashing, sort-merge, and nested loops, but did not consider nested-loops with index. The Tandem group [EGKS90] state that their NonStop SQL system uses a variant of parallel nested loops with index algorithm if the appropriate indices exist and one of the relations is small, and hashing followed by sort-merge otherwise, but did not compare the two algorithms. Wolf et al. [WDYT90, WDY90] consider the performance of parallel hashing and sort-merge algorithms in the presence of skew, but do not consider parallel nested loop algorithms.

In early work, Blasgen and Eswaran [BE77] compared sort-merge with nested-loops with index on uniprocessors, and concluded that sort merge almost always wins unless there is an appropriate clustered index on one of the join operands. Recently, Shekita and Carey [SC90] compared a number of uniprocessor join algorithms, including pointer based join algorithms, nested-loops with index, sort-merge, and hybrid hash. In the portion of that work relevant to this paper, they found that index nested loops works well if one of the relations is small. Valduriez [Val87] proposed the use of an auxiliary data structure called a *join index* in join processing, and showed that in many cases a join algorithm using a join index can out-perform hybrid hashing. Although they both use the term "index," the join index algorithm differs significantly from the nested loops with index algorithms. A join index essentially precomputes the join by storing pairs of tuple id's, one pair of tuple id's for each tuple that would appear in the join result. The indices used in nested-loop with index are just the usual B-tree indexes built by current relational DBMS. Omiecinski and Lin [OL89] consider the use of join indices in a parallel environment.

The remainder of the paper is organized as follows. We describe the hybrid-hash algorithm and four variants of nested loops with index in Section 2. Section 3 describes our analytic model while Section 4 describes results derived from the model. Section 5 describes our implementation and experiments in Gamma. Our conclusions are contained in Section 6.

2 Algorithms

Throughout this paper we will adopt the convention that the two relations being joined are R and S, and that the join condition is $R.A = S.B$. Furthermore, we will assume that there is an index on the attribute $S.B$. Then the uniprocessor nested loops with index join algorithm is just the following [BE77]:

```
for each tuple r in R do
  lookup the value r.A in the index on s.B;
  for each S tuple s returned by the lookup
    output answer tuple (r,s);
  endFor;
endFor;
```

We now consider how to implement a version of this algorithm in a shared-nothing parallel database system [Sto86]. When mapping this algorithm to a shared-nothing parallel system, one must first specify how the relations R and S are stored in the system. We will assume that both R and S are declustered throughout the processors of the system on attributes other than $R.A$ and $S.B$. That is, there is no *a priori* relationship between the value an R tuple r has in its A attribute and where that tuple r is stored in the system. Similarly, the location of an S tuple s is independent of the value that appears in $s.B$.

This immediately creates the problem of how to ensure that every R tuple r "meets" with every S tuple s such that $r.A = s.B$. We considered two mechanisms to guarantee this, *replication* and *mapping*; each is dealt with in a subsection below.

2.1 Replicating Parallel Nested Loops

The most obvious solution is to broadcast every R tuple to every site in the multiprocessor. After broadcasting R, every site in the multiprocessor joins R with its local fragment of S, using indexed nested loops to perform the local join. In high-level pseudocode, the algorithm is the following:

```
each processor p broadcasts its local R
    fragment to all other processors;
each processor p joins all of R with its local
    S fragment, using indexed nested loops;
```

We call this algorithm "replicating nested loops with index," and will abbreviate this to **RNL**. RNL is similar to the distributed database join algorithm known as *fragment-replicate*, originally proposed in [ESW78], with the "fragment" phase a no-op since this algorithm begins with S fragmented about the sites of the system. RNL is also the algorithm used by Tandem if the appropriate index exists, one of the relations is small, and the join is on a key for the small relation [EGKS90]. Recently Stamos and Young have proposed an improvement on the full fragment-replicate algorithm [SY89] that partially replicates and redistributes both input relations. While this algorithm improves on the network cost of a full fragment-replicate, it is not appropriate for use in RNL because it must redistribute portions of both R and S, which eliminates the possibility of using a preconstructed index on $S.B$.

2.2 Subsetting Parallel Nested Loops

The replicate nested loops algorithm has the disadvantage that many R tuples are shipped to sites at which there are no matching S tuples. In fact, in the case where every R tuple joins with at most one S tuple (as in a foreign key-key join), on a k processor system only $1/k$ of the R tuples sent to a site will actually find a matching S tuple. The subsetting parallel nested loops algorithm seeks to avoid this wasted effort, by determining for each R tuple r the subset of sites at which matching S tuples might reside. We will abbreviate "subsetting nested loops with index" by **SNL**.

The difficulty is that since we cannot assume any association between the $S.B$ attribute values and the partitioning attribute of S, there is no information in the system catalogs that can determine to which subset of sites a given R tuple should be sent. Our solution is to build an additional relation, which we will call *MapS*, that maps from S join attribute values to S partitioning attribute values. In more detail, suppose that S is range or hash partitioned [DGS+90] on attribute $S.P'$, where $P' \neq B$. Furthermore, let P be a key for S that contains P'. Then the relation *MapS* has the schema $MapS(P, B)$, with the semantics that there is a tuple m in *MapS* with $m.P = v_1$ and $m.B = v_2$ if and only if there is a tuple s in S with $s.P = v_1$ and $s.B = v_2$. The intention is that in a *MapS* tuple $(m.B, m.P)$, the attribute of $m.P$ is a surrogate for an S tuple. Since *MapS* has one tuple for every S tuple, in general *MapS* could be too large to replicate at all sites. For this reason *MapS* is declustered by hashing throughout the system on the attribute *MapS.B*. The concept of such a mapping relation from attributes to surrogates was introduced by Copeland and Khoshafian [CK85] in their decomposition storage model. In the remainder of this paper, for simplicity we will assume that P' is itself a key for S, so we can use $P = P'$.

Note that *MapS* is not a join index [Val87]. A join index contains an entry for every pair of tuples (r, s) such that r is from R and s is from S and r and s join. *MapS* merely pairs S join attribute values with S partitioning attribute values, independent of R.

Given this *MapS* relation, to find out to which subset of sites a given R tuple r should be mapped is a two-step process:

1. Send r to the site that contains *MapS* tuples m with $m.B = r.A$. Note that this is possible by consulting the system catalogs, since *MapS* is hash or range partitioned on $m.B$.

2. For each tuple m with $m.B = r.A$, send r to the site containing S tuples with $S.P = m.P$. Note that this too is possible to accomplish by consulting the system catalogs, since S is partitioned on the attribute $S.P$.

After R has been redistributed in this fashion, SNL proceeds like RNL after the broadcast: each site p uses nested-loops with index to join the subset of R mapped to p with the subset of S that is stored at p.

2.3 The Sorting Variants

We also investigated the effect of sorting the R tuples before joining them with S (in RNL and SNL) and also before looking up *MapS* tuples (in SNL.) We denote the resulting algorithms **RNL-S** and **SNL-S**. Sorting the outer relation is a well-known optimization for nested loops with index algorithms. If the S relation fragment at a site has a clustered index on the join attribute, by sorting the R tuples before joining them with S we can insure that the data and index pages of S are read only one in the join process. Similarly, if *MapS* is clustered on *MapS.B*, then by sorting incoming R tuples we can avoid re-reading data and index pages of *MapS*.

In the SNL-S algorithm, we sort the R tuples twice: first at each mapping site, second at each joining site.

The reason for this is as follows. Consider what happens at some processor p during the actual join of R with S (e.g., when p is receiving tuples from the mapping phase of the algorithm and joining them with S.) In general, processor p will see a stream of incoming R tuples from all other processors in the system. This stream will be "chunked" into a sequence of messages; while the tuples within each message will be sorted (if they were sorted at the mapping processor that sent that message to p), there is no guarantee that the tuples in the combined stream of the messages will be in sorted order, since there is no guarantee about the relationship between the tuples in a message from one processor and the tuples in a message from another. An optimization of this technique is discussed in Subsection 4.2.

2.4 Hybrid Hash

The hybrid hash join algorithm has been described in detail elsewhere in the literature [DKO+84, DG85, SD89]; here we give a top-level overview of the important aspects of parallel hash join algorithms. In the following we will denote hybrid hash by **HH**.

A basic parallel hash join of R and S with the join condition $R.A = S.B$ begins by redistributing both R and S. If the hash function chosen for the redistribution is $h(X)$, then a tuple r in R is sent to the processor determined by $h(r.A)$; similarly, a tuple s in S is sent to the processor determined by $h(s.B)$. Let the subset of R mapped to processor p_i be denoted R_i, and the subset of S mapped to p_i be denoted S_i. Then after redistributing R and S, each processor p_i joins the local fragments R_i and S_i. These local joins are performed by another use of hashing: some hash function $h'(X)$ is chosen and used to build an in-memory hash table of R tuples on the attribute $R.A$; then this in-memory hash table is probed for each S tuple s based on the value in the attribute $s.B$.

HH differs from this basic hash algorithm in that it attempts to keep R tuples in memory after the redistribution stage, so that these tuples don't have to be re-read from disk in the join phase. In more detail, if some fraction f of R can be retained in memory, then only $(1-f)$ of R is written to disk at the joining processor. Equally importantly, only $(1-f)$ of S needs to be written to disk at the joining processor, since tuples of S that join with the memory-resident portion of R are joined "on the fly" then discarded.

2.5 Other Alternatives

The parallel nested loop algorithms and the hybrid hash algorithm can both be viewed as consisting of two somewhat orthogonal subtasks:

1. How to redistributed the relations.

   Parallel hybrid hash uses hash partitioning to redistribute both input relations, while the parallel nested-loops algorithms redistribute only one of the relations (either replicating the relation or subsetting it.)

2. How to do the local joins after redistribution.

   Parallel hybrid hash uses hybrid hash to do the local joins, while parallel nested loops with index uses the nested loops with index algorithm.

From this perspective, it is clear that there are two other classes of algorithms to consider: hashing-redistribution of both relations followed by indexed nested loops within sites, and replicating/mapping one of the relations followed by hybrid hash within the sites. We did not explore these algorithms because they are unlikely to perform well.

Hash-based redistribution followed by indexed nested loops is unlikely to perform as well as parallel hybrid hash because the cost of creating a clustered index (which must be done at query evaluation time) after redistribution will be higher than the cost of building a hash table. Broadcast/mapping redistribution of one of the relations followed by hybrid hashing is unlikely to perform as well as the parallel nested loop with index algorithms because it ignores a pre-existing index on the join attribute, hence must completely process both relations in the local joins at each site after redistribution.

3 An Analytic Model

In this section we develop an analytic model intended to predict the performance of the join algorithms. The purpose of this model is not so much to predict absolute performance as it is to allow us to identify the important trends and characteristics in the *relative* performances of the algorithms. We consider each algorithm in turn. In each case, we separate the cost of the algorithm into I/O, CPU, and network. In each case, we omit the cost of writing the answer relation, since this cost is the same for all algorithms. However, we do include the cost for reading the input relation(s), since this cost differs between the algorithms. Furthermore, we assume that messages each contain one page. We also make the simplifying assumption that the running time of the parallel algorithm can be calculated by computing the time required for a single site in isolation to complete all of the tasks required by the algorithm. The model takes as input the following parameters describing the input relations:

$|R|$ number of pages in R

$\|\|R\|\|$	number of tuples in R
$\|S\|$	number of pages in S
$\|\|S\|\|$	number of tuples in S

Furthermore, except where explicitly noted otherwise we assume that the join is foreign key-key, meaning that each R tuple joins with exactly one S tuple. We assume that each R tuple r joins with a randomly chosen tuple of S; this means that it is possible that two R tuples join with the same S tuple.

The following parameters describe the hardware configuration:

procs	number of processors
mem	number of memory pages
io	time for an I/O (read or write)
msg	time to send/receive a message
hash	time to compute hash function
insert	time to insert in hash table
probe	time to probe hash table

3.1 Hybrid Hash

A critical parameter to the performance of parallel hybrid hash is the percentage of R that fits in memory, since this percentage of R and S can be processed without ever writing the tuples to disk after repartitioning. In the model below we refer to this fraction of R as "part0frac," for "partition zero fraction."

$$\begin{aligned}\text{HH}_{\text{io}} = \;& \text{// read local } R \text{ partition} \\ & (|R|/\text{procs}) * \text{io} + \\ & \text{// write overflow } R \\ & (|R|/\text{procs}) * (1 - \text{part0frac}) * \text{io} + \\ & \text{// read local } S \text{ partition} \\ & (|S|/\text{procs}) * \text{io} + \\ & \text{// write overflow } S \\ & (|S|/\text{procs}) * (1 - \text{part0frac}) * \text{io}\end{aligned}$$

The network cost of hybrid hash is given by

$$\begin{aligned}\text{HH}_{\text{net}} = \;& \text{// send } R \\ & (|R|/\text{procs}) * \text{msg} + \\ & \text{receive } R \\ & (|R|/\text{procs}) * \text{msg} + \\ & \text{// send } S \\ & (|S|/\text{procs}) * \text{msg} + \\ & \text{// receive } S \\ & (|S|/\text{procs}) * \text{msg}\end{aligned}$$

Finally, the CPU cost of hybrid hash is given by

$$\begin{aligned}\text{HH}_{\text{CPU}} = \;& \text{// find destination} \\ & (|R|/\text{procs}) * \text{hash} + \\ & \text{// insert in hash table} \\ & (|R|/\text{procs}) * \text{insert} + \\ & \text{// lookup in hash table} \\ & (|S|/\text{procs}) * \text{probe}\end{aligned}$$

This is intended to cover the CPU costs that are not covered by the expressions for the IO and network times. The CPU cost is probably the least accurate in this model; fortunately for the accuracy of the model, it is also by far the smallest component of the total cost of the algorithm.

3.2 Replicating Nested Loops (RNL)

Coming up with an expression for the running time of the nested loop with index algorithms is slightly more complex than was the case for hybrid hash because the total number of I/O's is not determined by the algorithm (as is the case for hybrid hash) but is determined by the combination of the algorithm, the buffer pool size, and the buffer replacement policy. For the replicating nested loops with index algorithm we assume that the buffer pool pages are allocated with the following decreasing priority:

- Index pages of S;
- Data pages of S;
- Data pages of R.

The motivation for this policy is that (1) if S and/or S index pages cannot be memory resident for the duration of the algorithm, they could be read multiple times, whereas if an R page does not fit in memory it need only be spooled to disk (when it arrives at the join site) and re-read once (when the tuples on the page are actually joined with S); (2) the index on S is in general smaller than S, so keeping the index in memory "costs" fewer buffer pages than keeping S in memory. The priority of S index pages over S data pages also roughly approximates the behavior of the algorithm under an LRU buffer management policy, since the index pages can be expected to be hotter than the S data pages.

With this in mind we can now specify the cost of the replicating nested loops algorithm. The following formula uses the quantities indexReads, SReads, and RReads. These quantities will be defined after the formula.

$$\begin{aligned}\text{RNL}_{\text{IO}} = \;& \text{initRReads} + \text{indexReads} + \\ & \text{SReads} + \text{RJoinReads}) * \text{io}\end{aligned}$$

InitRReads is just $|R|/\text{procs}$, for the read of the original fragments of R. (Recall that since RNL replicates all of R everywhere, every processor must process all of R.)

To calculate the number of index page reads, we assume that initially there are no index pages in the buffer

pool. Then at any given time during the execution of the algorithm, the probability that a given R tuple "hits" in the buffer pool is given by the ratio of the number of index pages in memory to the total number of index pages. If an R tuple "misses" in the buffer pool, and there are still available buffer frames, then the number of resident index pages is incremented.

The calculation for the number of S page reads is identical except that instead of all memory being initially available to S, only the memory left over after fitting the S index into memory (if any) is available to S. Finally, the number of RJoinReads is calculated as follows: first define the number of R overflow pages to be $|R|$ minus the number of buffer pool pages available after fitting both S and the index on S in memory. Then there is one read and one write for each R overflow page; there are no join reads for R pages that are not overflow pages, since these pages remain in the buffer pool throughout the join.

The network cost of the replicating nested loops algorithm is easy to calculate:

$$\begin{aligned}\text{RNL}_{\text{net}} = \;& \text{// broadcast } R \text{ fragment} \\ & |R| * \text{msg} + \\ & \text{// receive all of } R \\ & |R| * \text{msg}\end{aligned}$$

Again, each processor must send and receive $|R|$ messages (rather than $|R|/procs$) because RNL fully replicates R. That is, we are assuming that for a site to broadcast $|R|/\text{procs}$ pages requires sending a total of $|R|/\text{procs} * \text{procs} = |R|$ pages.

Finally, the CPU cost of replicating nested loops is given by

$$\begin{aligned}\text{RNL}_{\text{CPU}} = \;& \text{// cost for index probes} \\ & \|R\| * \text{CPUIndex}\end{aligned}$$

As in the case of hybrid hash, this is the CPU cost unaccounted for in the IO and network costs. Also as in the model for hybrid hash, the CPU cost is a small fraction of the total cost.

3.3 Replicating Nested Loops with Sorting (RNL-S)

If the index on the join attribute in S is a clustered index, RNL can be sped up by sorting R before joining it with S. In this case the total number of S reads is just the number of S pages that contain at least one tuple that joins with some R tuple. We calculate this quantity, *numSReads*, by assuming that the tuples that join with the R tuples are randomly distributed throughout the S pages. From this viewpoint, this is just a classical "balls in bins" problem from elementary probability theory. The number of S index reads, *numSIndexReads*, are computed in an analogous way. The number of R reads after the initial scan of R is just one read for the join (assuming that the sort leaves R on disk; Subsection 4.2 discusses an optimization of SNL-S that avoids much of the I/O cost of this sort) plus the required number of reads and writes to sort R. In our analytic model we assume that R can be sorted in two passes (given modern memory sizes, even a 100GByte relation can be sorted in two passes [STG+90]). This means that the I/O cost for replicating nested loops with sorting is given by

$$\begin{aligned}\text{RNL-S}_{\text{io}} = \;& \text{// scan initial } R \text{ fragment} \\ & (|R|/\text{procs}) * \text{io} + \\ & \text{// two-pass sort of } R \\ & 4 * |R| * \text{io} + \\ & \text{// } R \text{ reads for actual join} \\ & |R| * \text{io} + \\ & \text{// } S \text{ reads} \\ & \text{numSReads} * \text{io} + \\ & \text{// } S \text{ index reads} \\ & \text{numSIndexReads} * \text{io}\end{aligned}$$

The network cost is identical to the network cost of the replicate nested loops without sorting. The CPU cost for the replicate nested loops is just that of the replicate nested loops, plus additional CPU for the sort. In our analytical model, we used the formula from in [DKO+84]:

$$\|R\| * (\log \|R\| * \text{keyswap} + \text{move})$$

3.4 Subsetting Nested Loops (SNL)

Recall that in SNL, instead of broadcasting R to all processors, each R tuple r is first sent to an intermediate processor for a lookup on the *MapS* relation, then forwarded to all processors that have S tuples that join with r. For this reason, in the subsetting nested loops algorithm we need to specify the number of S tuples that join with a given R tuple. In the model, we assume that this number is a constant, the *fanout*. Furthermore, we assume in the model that if an R tuple joins with k tuples of S, those k tuples of S are located on k distinct processors.

As with RNL, the number of reads for the SNL depends upon the interaction between the algorithm and the buffer pool. At the top level, the total I/O time is given by the expression

$$\begin{aligned}\text{SNL}_{\text{io}} = & (\text{initRReads} + \text{mapSReads} + \text{RMapReads} + \\ & \text{indexReads} + \text{SReads} + \text{RJoinReads}) * \text{io}\end{aligned}$$

where initRReads, indexReads, SReads, and RJoinReads have the same meaning as in RNL, mapSReads are the number of IOs for *SMap* pages, and RMapReads is the number of IOs during the mapping of R tuples to final destination processors.

To calculate all of these read quantities we again need to decide upon the priority for buffer pool pages. If we assume that the mapping and joining phases of the algorithm are sequential (the join of R and S does not begin until all sites have finished mapping R tuples to final destination sites) then there is no contention between *MapS* pages and S index pages. In the model we assume that *MapS* is stored like a B-tree index (with $< S.B, S.P >$ entries instead of $< s.B, \text{rec. id} >$ pairs in the leaf pages), and that all but the leaf pages of the index are memory resident. The MapSReads are calculated exactly as the S index reads were calculated in RNL. (That is, by assuming that a map lookup hits the buffer pool with probability given by the ratio of the current number of *MapS* pages in the buffer pool to the total size of *MapS*.) The RMapReads are calculated by first defining the R overflow pages to be $|R|/\text{procs} - (\text{memory} - \text{MapSPagesResident})$, and charging one read and one write for each R overflow page.

The quantities initRReads, indexReads, SReads, and RJoinReads are all calculated exactly as in RNL, except that now the number of R tuples per processor is $(\|R\| * \text{fanout})/\text{procs}$ instead of $\|R\|$, and the number of R pages per processor is $(|R| * \text{fanout})/\text{procs}$ instead of $|R|$.

The network cost of the subsetting nested loops is given by the expression

$$\begin{aligned}\text{SNL}_{\text{net}} = \;& \text{// distribute } R \text{ for mapping} \\ & ((|R| * \text{fanout})/\text{procs}) * \text{msg} + \\ & \text{// receive } R \text{ for mapping} \\ & ((|R| * \text{fanout})/\text{procs}) * \text{msg} + \\ & \text{// distribute } R \text{ for joining} \\ & ((|R| * \text{fanout})/\text{procs}) * \text{msg} + \\ & \text{// receive } R \text{ for joining} \\ & ((|R| * \text{fanout})/\text{procs}) * \text{msg}\end{aligned}$$

Finally, the CPU cost of subsetting nested loops is given by

$$\begin{aligned}\text{SNL}_{\text{CPU}} = \;& \text{// cost for index probes} \\ & 2 * (\|R\|/\text{procs}) * \text{CPUIndex}\end{aligned}$$

where the factor of two arises because there is one index lookup for the *MapS* lookup, and another for the index lookup in the actual join.

3.5 Subsetting Nested Loops with Sorting (SNL-S)

As with the RNL, in the presence of a clustered index on $S.B$ the number of IOs can be dramatically reduced by sorting R before joining. The same technique can be used to reduce the number of IOs in the mapping phase if *MapS* is clustered on *MapS.B*. In this subsection we consider this case.

The IO cost for SNL-S is given by

$$\begin{aligned}\text{SNL-S}_{\text{io}} = \;& \text{// scan initial } R \text{ fragment} \\ & (|R|/\text{procs}) * \text{io} + \\ & \text{// two-pass sort of } R \text{ at map site} \\ & 4 * (|R|/\text{procs}) * \text{io} + \\ & \text{// } R \text{ reads for mapping} \\ & (|R|/\text{procs}) * \text{io} + \\ & \text{// } \textit{MapS} \text{ reads} \\ & \text{numSMapReads} * \text{io} + \\ & \text{// two-pass sort of } R \text{ at join site} \\ & 4 * ((|R| * \text{fanout})/\text{procs}) * \text{io} + \\ & \text{// } R \text{ reads for join} \\ & ((|R| * \text{fanout})/\text{procs}) * \text{io} + \\ & \text{// } S \text{ index reads} \\ & \text{numSIndexReads} * \text{io} + \\ & \text{// } S \text{ reads for join} \\ & \text{numSReads} * \text{io}\end{aligned}$$

Here numSMapReads is calculated in an analogous fashion to the way numSIndexReads is calculated in the replicating nested loops with sorting algorithm; the quantities numSIndexReads and numSPageReads are calculated in exactly the same way as in the replicating nested loops with sorting algorithm, only now the number of R tuples per processor is $\|R\| * \text{fanout}/\text{procs}$ instead of $\|R\|$.

The network cost is identical to that of SNL. Finally, the CPU cost for the sort at a mapping processor is

$$(\|R\|/\textit{procs}) * (\log(\|R\|/\textit{procs}) * \text{keyswap} + \text{move})$$

while the CPU cost for the sort at a join processor is

$$(\|R\|/\textit{procs}) * \text{fanout} * (\log((\|R\|/\textit{procs}) * \text{fanout}) * \text{keyswap} + \text{move})$$

4 Analytic Model Experiments

In this section we present results from experiments with the analytical model. In all cases, we fixed $\|S\|$ at 30K pages, each of 50 tuples, for 1.5M tuples. We also fixed the number of R tuples per page at 50, and varied the number of pages in R. Also except where noted otherwise, we assumed that the fanout (average number of S tuples that each R tuple joins with) was one. The hardware parameters we used were:

comp	0.010	// msec to compare keys
keyswap	0.030	// msec to swap a pair of keys
hash	0.010	// msec to hash a key
tuplemove	0.053	// msec to move a tuple in mem
swap	0.150	// msec to swap two tuples
io	30.000	// msec to do sequential IO
msg	6.000	// msec to send a message

These costs approximate costs that we have measured in Gamma. Except where noted otherwise, we have assumed a 30 processor system.

4.1 Memory Size

Memory size is a critical parameter to the performance of all of the algorithms. In this section, we compare the algorithms at three memory sizes:

1. Small. Here none of R, S, the index on S, or *MapS* fit in memory.
2. Medium. Here the indexes on S and *MapS* fit entirely in memory, but neither R nor S themselves fit in memory.
3. Large. Here all of R, S, the index on S, and *MapS* fit in memory.

In each case, the basic experiment is to fix the size of S and vary the size of R, plotting the performance of the algorithms as a function of the size of R.

Figure 1 shows the analytic model performance for all five algorithms in the small memory case (50 memory pages per processor). The graph illustrates that when memory is scarce, of the five algorithms only HH and SNL-S are reasonable alternatives. Both the RNL and SNL have terrible performance since they do multiple I/Os per R tuple. RNL-S also has poor performance, primarily due to the cost of writing and reading all of R at every processor and the cost of broadcasting R to every processor in the system.

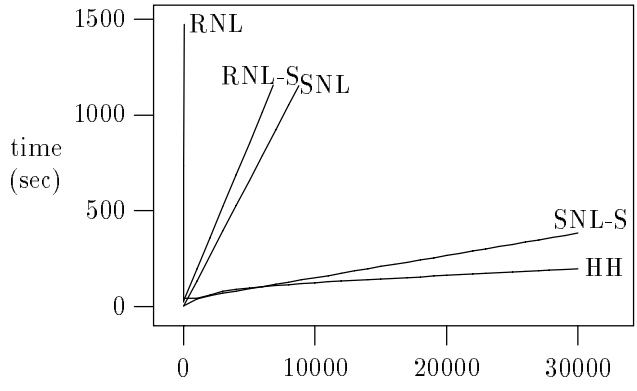

Figure 1: Analytic model performance of all algorithms, small memory case.

Table 1 gives the rough breakdown on the running times of the algorithms into CPU, IO, and network costs. The numbers in that table are the ranges (minimum to maximum) of the fractions over all data points in the graphs in Figure 1. In all cases the model predicts that CPU costs are a small fraction of total cost, although the sorting versions of the algorithms have considerably higher CPU costs than their non-sorting counterparts.

Algorithm	frac. CPU	frac. network	frac. IO
RNL	< 0.01	< 0.01	> 0.99
RNL-S	$0.02 - 0.20$	$0.01 - 0.07$	$0.75 - 0.96$
SNL	< 0.01	< 0.01	> 0.99
SNL-S	$0.01 - 0.14$	$0.00 - 0.06$	$0.80 - 0.99$
HH	< 0.01	$0.12 - 0.29$	$0.71 - 0.87$

Table 1: Breakdown of execution times, small memory case.

The network costs are a significantly higher fraction of HH than of the other algorithms; this is partially because HH must redistribute both R and S, and partially because the other costs of HH (CPU and IO) are low when compared with those costs for the other algorithms.

Figure 2 shows the performance of the best algorithms from Figure 1 in the interesting region of the comparison (where HH and SNL-S are roughly comparable.) Figure 2 makes a point that is consistent throughout our experiments: SNL-S only beats HH over a small portion of the total problem space, but in this region, SNL-S can be much faster than HH.

The "knee" in the graph of HH occurs when R no longer fits entirely in memory. From this point on, every additional R page causes $1.0+(|S|/|R|)$ I/Os for overflow handling. As $|R|$ grows this quantity shrinks, and the slope of the curve for HH diminishes. The curve for SNL-S in Figure 2 begins with a steep slope, because for very small R relations, the number of tuples in R is less than the number of pages in S, and essentially every additional R tuple results in an additional I/O on an S page. When the number of R tuples is approximately equal to the number of pages in S, additional R tuples do not cause any additional S I/Os, so the only increase in I/O time is due to the additional fractional I/O for reading the tuples on the new R pages. At this point the slope in the curve for SNL-S diminishes.

Figure 3 shows the performance of all the algorithms in the medium memory configuration (here, 200 pages per processor.) RNL and SNL perform better here, while the effect of more memory on RNL-S and SNL-S is less pronounced. This is because RNL and SNL are more directly impacted by the fraction of *SMap* and the index on *S.B* that fits in memory.

Figure 4 shows SNL-S and HH on the medium memory case. This graph shows that SNL-S is not able to effectively make use of the additional memory, whereas HH is able to use the memory to avoid having to spool overflow tuples to disk, hence here HH beats SNL-S by a wider margin.

Figure 5 gives the analytic model performance for all algorithms when the memory is large (2150 pages per

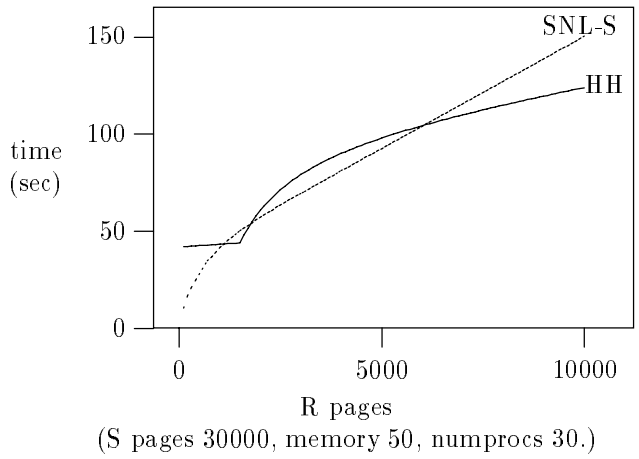

Figure 2: Analytic model performance of SNL-S and HH, small memory case.

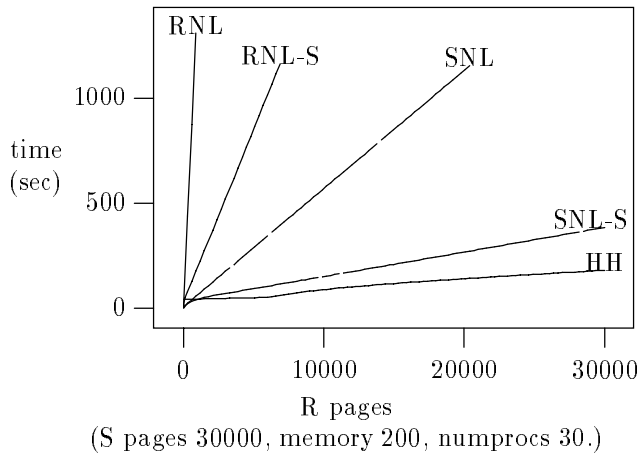

Figure 3: Analytic model performance of all algorithms, medium memory case.

processor). Here note that the sorting variants of the nested loops algorithms actually perform worse than the non-sorting variants. This is because the memory is now large enough to hold S, the index on S, and *SMap* in memory, so the non-sorting variants of the algorithms just fault these relations and indices into memory then do no more I/O, hence every page is read at most once, just as in the sorting variants of the algorithms. In this case the sorting overhead is wasted.

Figure 6 focuses on the SNL-S, SNL, and HH algorithms in the large memory case. Again, while the nested loops algorithms only beat HH for small instances of R, they can be much faster.

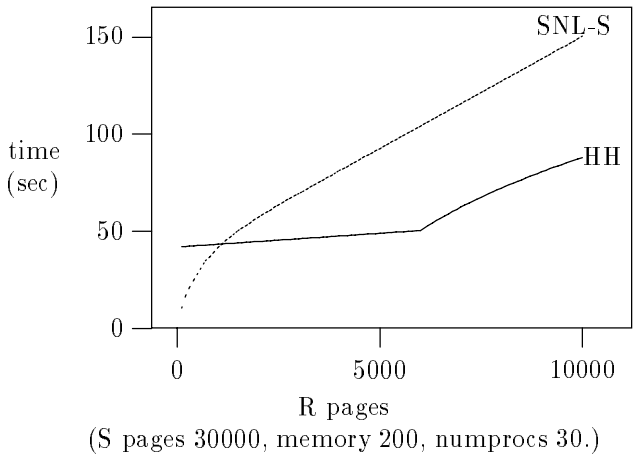

Figure 4: Analytic model performance of SNL-S and HH, medium memory case.

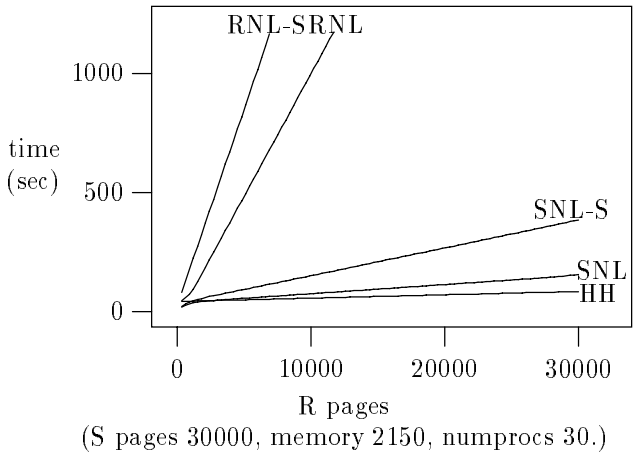

Figure 5: Analytic model performance of all algorithms, large memory case.

4.2 An Optimization for SNL-S

The sorting variants of the algorithms as described in Section 2 can be improved with the following sorting optimization: instead of a two-pass sort that begins and ends with the relation on disk, the R tuples can be sorted into runs as they come off the network, then written in sorted runs. Next, these sorted runs can be merged, but instead of writing the result of the merge back to disk, the tuples produced by the merge can be joined with *MapS* and S "on the fly." In this way R is sorted without incurring any overhead beyond that required to spool R to disk and re-read it in the join. Figure 7 shows SNL-S, OSNL-S (Optimized SNL-S), and HH for the medium memory configuration. While the optimization clearly improves upon the performance of SNL-S, it does not change the overall result: the nested loops algorithms

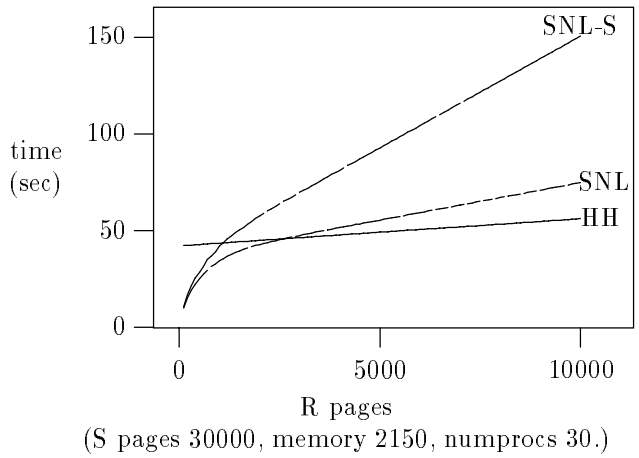

Figure 6: Analytic model performance of SNL, SNL-S and HH, large memory case.

do well only when R is very small, but they do very well in that case.

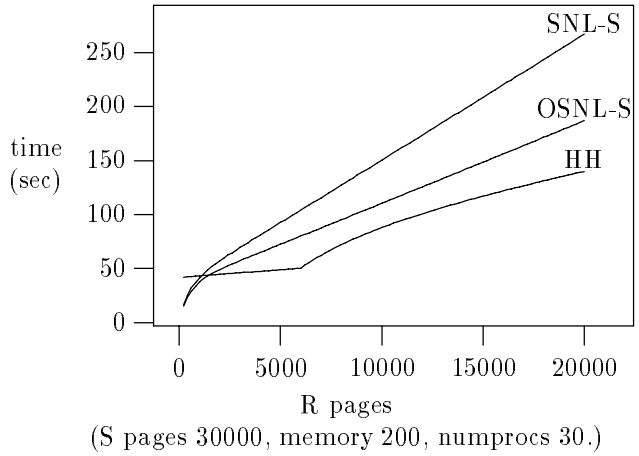

Figure 7: Optimized versus basic SNL-S.

5 Implementation and Experiments

In this section we describe the implementation and experiments with the nested loop join algorithms in Gamma. Our goal was to explore the performance of the algorithms in an implementation, and also to investigate how naturally the nested loops algorithms could be implemented in a system that already has indices and the hash-join algorithms available.

5.1 Implementation

Gamma falls into the class of *shared-nothing* [Sto86] architectures. The hardware consists of a 32 processor Intel iPSC/2 hypercube. Each processor is configured with a 80386 CPU, 8 megabytes of memory, and a 330 megabyte MAXTOR 4380 (5 1/4 in.) disk drive. Each disk drive has an embedded SCSI controller that provides a 45 Kbyte RAM buffer that acts as a disk cache on sequential read operations. The nodes in the hypercube are interconnected to form a hypercube using custom VLSI routing modules. Each module supports eight full-duplex, serial, reliable communication channels operating at 2.8 megabytes/sec.

Gamma is built on top of an operating system designed specifically for supporting database management systems. NOSE provides multiple, lightweight processes with shared memory. A non-preemptive scheduling policy is used to help prevent convoys [BGMP79] from occurring. NOSE provides communications between NOSE processes using the reliable message passing hardware of the Intel iPSC/2 hypercube. File services in NOSE are based on the Wisconsin Storage System (WiSS) [CDKK85].

The services provided by WiSS include sequential files, byte-stream files as in UNIX, B^+ tree indices, long data items, an external sort utility, and a scan mechanism. A sequential file is a sequence of records that may vary in length (up to one page) and that may be inserted and deleted at arbitrary locations within a file. Optionally, each file may have one or more associated indices that map key values to the record identifiers of the records in the file that contain a matching value. One indexed attribute may be designated to be a clustering attribute for the file.

The basic code that needed to be added to Gamma included:

1. The code to perform a local indexed nested loops join.

2. The code to broadcast tuples to multiple sites.

3. The code to do the *SMap* lookup and redistribution for the SNL and SNL-S algorithms.

Item one was straightforward. Item two was necessary because the tuple redistribution mechanisms used by the parallel hybrid-hash algorithm assume that each tuple is sent to exactly one destination. Fortunately, we had already added code to do the broadcast and subset redistributions in our work on skew-handling join algorithms [DNSS92], so no new code needed to be written. Items one and two were all that was necessary to implement the RNL and RNL-S algorithms.

Item three was less straightforward. The difficulty is that although *SMap* is conceptually a lot like an index, it is used in a very different manner: instead of using values to associatively access tuples, it uses join attribute values to determine a set of target processors. Accordingly, to implement *SMap* as described in Section 2 would have required a rewrite of a significant portion of the system code that implements indices. To avoid this rewrite, instead of making *SMap* an access method (index), we made it a relation. In more detail, $SMap(P, B)$ is a binary relation containing a tuple $(m.P, m.B)$ for every tuple $(s.P, s.B, \ldots)$ in S (recall that P is the partitioning attribute of S). We store *SMap* hash partitioned on *SMap.B*.

We then noticed that if the optimizer rewrites the binary join

```
range of r is R
range of s is S
retrieve (r.all, s.all)
where r.A = s.B
```

into the three relation join

```
range of r is R
range of m is SMap
range of s is S
retrieve (r.all, s.all)
where r.A = m.B and m.P = s.P
```

then the result of this join is the same as the result of the two relation join. (Recall that by definition $S.P$ is a key for S.) More importantly, after some minor modifications to the Gamma scheduler, when evaluating this three way join, Gamma does almost exactly the SNL join algorithm. The execution works as follows:

1. Each R tuple r is shipped to the processor that contains the *SMap* tuples m with $m.B = r.A$.

2. At each processor, the fragment of R received in Step 1 is joined with the local fragment of *SMap*. The tuples produced include the attributes A (from R), and B and P (from m).

3. Each tuple j produced in the join in Step 2 is sent to the processor that contains the S tuples s with $s.P = j.P$.

4. At each processor, the tuples sent to proc_j in Step 3 are joined with S.

The main difference between the algorithm this produces and the SNL algorithm we described in Section 2 is that the join between R and *SMap* in Step 2 is not just a lookup on *SMap*. This join is accomplished by a (local) nested loop with index join algorithm; we created a clustered index on *SMap.B* to make this efficient.

To implement RNL-S and SNL-S, we used existing system sorting code. This code assumes that the input relation to be sorted is on disk both at the beginning and end of the sort, so we implemented these algorithms without the optimization described in Subsection 4.2.

5.2 Experiments

We configured the system to use 125 buffer pool pages per processor, each of 8K bytes, and used 30 processors. We also used 8K byte network packets. In all cases, the tuple size of both R and S was 208 bytes. We performed two sets of experiments with the implementation, roughly corresponding to the "medium memory" and the "large memory" cases from the analytic model.

Figure 8 contains the results of an experiment in S contained 500K tuples. At approximately 40 tuples per page, this is approximately 12,500 pages. Spread over 30 processors, this is roughly 400 pages per processor. This will not fit entirely in the 125 page buffer pool, although the $S.B$ index and *SMap* will fit. We varied the size of R from 10 to 100K tuples. The relative performance of the algorithms closely matches that predicted by the analytic model in Figure 3.

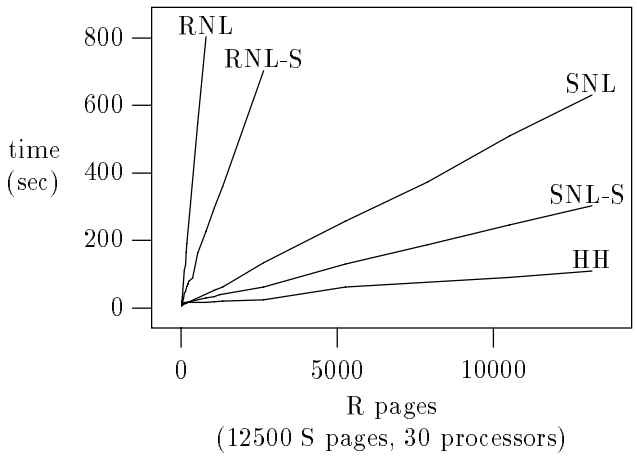

Figure 8: Experimental data from Gamma, medium memory case.

Figure 9 shows the Gamma performance of HH and SNL-S in the medium memory case. The graph is roughly analogous to Figure 4; in Figure 9 since there is relatively more free memory, HH does not have to use multiple partitions and its performance does not degrade as in Figure 4.

For the large memory case, because we were limited by the availability of physical memory, instead of growing the buffer pool, we shrank the relations, setting S to contain 100K tuples. With 8K pages, the entire relation occupies about 2500 pages. Spread over 30 processors,

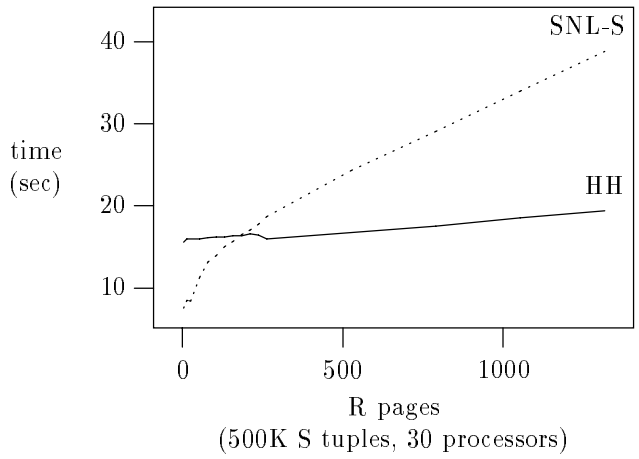

Figure 9: Experimental data from Gamma, medium memory case.

this gives roughly 80 pages per processor, which fits in the 125 page buffer pool with sufficient free space to also hold the index pages and *SMap* pages. We varied the size of R from 10 to 100K tuples. Figure 10 shows the results from this experiment. This graph corresponds to the graph in Figure 5 generated by the analytic model.

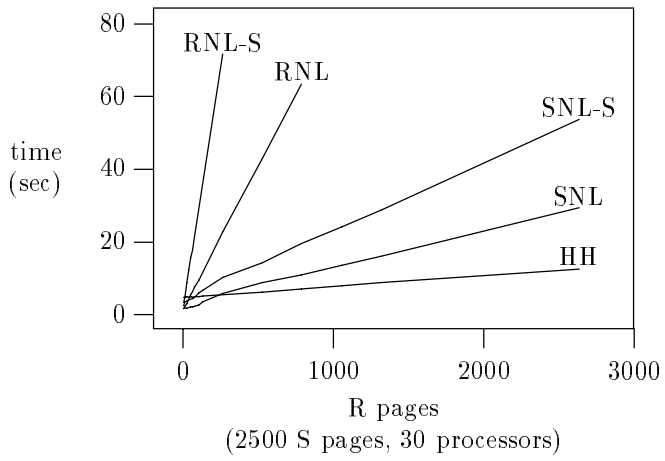

Figure 10: Experimental data from Gamma, large memory case.

Finally, Figure 11 shows the performance of SNL, SNL-S, and HH in the large memory case. As predicted by the analytic model (see Figure 6), SNL beats SNL-S in this region, and both SNL and SNL-S beat HH for very small R relations.

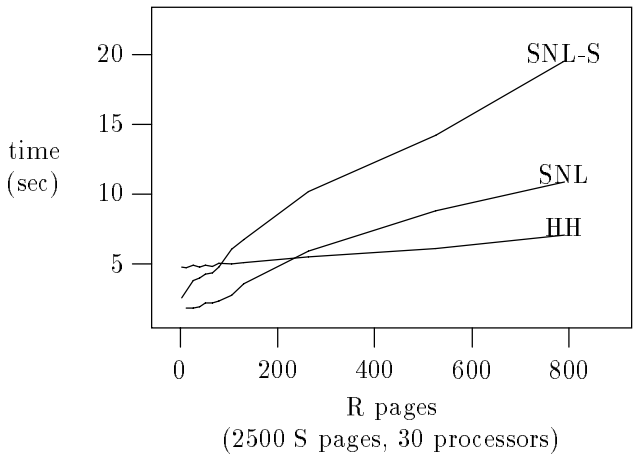

Figure 11: Experimental data from Gamma, large memory case, SNL, SNL-S, and HH.

6 Conclusion

Our analytic and experimental investigation of parallel hybrid hashing vs. parallel nested loops with index confirms the intuition that while hybrid hashing out performs nested loops for most combinations of input relation sizes, if the relation sizes are very different, then nested loops with index provides significantly better performance than hybrid hashing. Furthermore, of the nested loops algorithms we considered (SNL, SNL-S, RNL, RNL-S), only SNL-S provided acceptable performance in general, although if the available memory is larger than the indexed relation, SNL is the algorithm of choice.

Since joins of relations of disparate sizes are not uncommon (e.g., a select-join where the selectivity is high), parallel database systems could profit from implementing both algorithms. However, having only the nested loops with index algorithm is not sufficient, since the nested loops algorithms only perform well when the appropriate index exists and the join input relation sizes are sufficiently different. Our simple cost formulas from the analytic model could be used by an optimizer to determine which algorithm to use.

References

[BE77] M. W. Blasgen and K. P. Eswaran. Storage and access in relational databases. *IBM Systems Journal*, 16(4), 1977.

[BGMP79] M. W. Blasgen, J. Gray, M. Mitoma, and T. Price. The convoy phenomenon. *Operating System Review*, 13(2), 1979.

[CDKK85] H-T. Chou, D. J. Dewitt, R. H. Katz, and A. C. Klug. Design and implementation of the Wisconsin Storage System. *Software—Practice and Experience*, 15(10):943–962, October 1985.

[CK85] G. Copeland and S. Khoshafian. A decomposition storage model. In *Proc. of the ACM SIGMOD Conference*, pages 268 – 279, Austin, Texas, May 1985.

[DG85] D. M. DeWitt and R. Gerber. Multiprocessor hash-based join algorithms. In *Proc. of the Twelfth VLDB*, pages 151–164, Stockholm, Sweden, 1985.

[DGS+90] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE TKDE*, 2(1), March 1990.

[DKO+84] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proc. of the ACM SIGMOD Conference*, pages 1–8, June 1984.

[DNSS92] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proc. of the Nineteenth VLDB*, Vancouver, British Columbia, August 1992.

[EGKS90] S. Englert, J. Gray, T. Kocher, and P. Shah. A benchmark of NonStop SQL Release 2 demonstrating near-linear speedup and scaleup on large database. In *Proc. of the SIGMETRICS Conference*, pages 245–247, May 1990.

[ESW78] R. Epstein, M. Stonebraker, and E. Wong. Distributed query processing in a relational database system. In *Proc. of the ACM-SIGMOD Conference*, 1978.

[KTMo83] M. Kitsuregawa, H. Tanaka, and T. Motooka. Application of hash to data base machine and its architecture. *New Generation Computing*, 1(1), 1983.

[OL89] R. Omiecinski and E. T. Lin. Hash-based and index-based join algorithms for cube and ring connected multicomputers. *IEEE TKDE*, 1(3):329–343, September 1989.

[SC90] E. J. Shekita and M. J. Carey. A performance evaluation of pointer-based joins. In *Proc. of the ACM-SIGMOD Conference*, pages 300–312, Atlantic City, New Jersey, May 1990.

[SD89] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proc. of the ACM-SIGMOD Conference*, pages 110–121, Portland, Oregon, June 1989.

[STG+90] B. Salzberg, A. Tsukerman, J. Gray, S. Uern, and B. Vaughan. FastSort: A distributed single-input single-output external sort. In *Proc. of the ACM-SIGMOD Conference*, pages 94–101, Atlantic City, New Jersey, May 1990.

[Sto86] M. Stonebraker. The case for shared nothing. *Database Engineering*, 9(1), 1986.

[SY89] J. W. Stamos and H. C. Young. A symmetric fragment and replicate algorithm for distributed joins. TR RJ 7118 (67667), IBM Research Division, Almaden Research Center, San Jose, California, December 1989.

[Val87] P. Valduriez. Join indices. *ACM TODS*, 12(2):218 – 246, June 1987.

[VG84] P. Valduriez and G. Gardarin. Join and semijoin algorithms for a multiprocessor database machine. *ACM TODS*, 9(1):133–161, March 1984.

[WDY90] J. L. Wolf, D. M. Dias, and P. S. Yu. An effective algorithm for parallelizing sort merge joins in the presence of data skew. In *Proc. of the Second ISDPDS*, pages 103–115, Dublin, Ireland, July 1990.

[WDYT90] J. L. Wolf, D. M. Dias, P. S. Yu, and J. J. Turek. An effective algorithm for parallelizing hash joins in the presence of data skew. IBM T. J. Watson Research Center Tech Report RC 15510, 1990.