HASH JOIN PROCESSING ON SHARED MEMORY MULTIPROCESSORS

Ambuj Shatdal
Jeffrey F. Naughton

Technical Report #1299

February 1996

Hash Join Processing on Shared Memory Multiprocessors

Ambuj Shatdal Jeffrey F. Naughton
Computer Sciences Department
University of Wisconsin-Madison

Computer Sciences Technical Report # 1299

February, 1996

Abstract

While most scalable database systems are designed for a shared nothing architecture, advances in hardware technology suggest that in the near future shared memory multiprocessors (SMPs) will be capable of handling all but the largest applications. This raises important questions about how scalable database systems should be architected; in particular, do SMPs require specially targeted algorithms, or will algorithms developed for shared nothing hardware suffice? As a first step towards answering the question, with an implementation on an SGI Challenge multiprocessor, we investigate the performance of hash join algorithms on SMPs. We first show that the shared nothing approach, if ported to an SMP by using a transparent message passing library, yields poor performance. However, we show that by using an optimized message passing library and modifying the join code so that it does not reuse message buffers, shared nothing algorithms become viable on an SMP. Next we study the performance of a commonly proposed simple parallel join algorithm designed for shared memory. We show that it performs only marginally better than the optimized shared nothing approach. Finally, we show that further performance improvements are possible over the simple shared memory algorithm by employing optimizations designed to maximize processor cache hits and minimize memory coherence traffic.

1 Introduction

There is an increasing interest in using shared memory multiprocessors, henceforth called SMPs, for high performance database systems. Since the scale of SMPs lies between those of uniprocessors and the shared nothing large scale parallel machines, it makes them suitable for all but the largest systems. They can handle this range effectively because they offer superior cost/performance as marginal performance gain in terms of parallelism and throughput due to addition of a CPU tends to be slightly higher than the cost of the hardware. Also, the shared address space allows natural extension of uniprocessor algorithms into the parallel domain making the SMPs easier to use than other parallel platforms. Finally, even without the technical

arguments for SMPs, the reality is that today most parallel machines sold are SMPs.

Today, however, most scalable parallel database systems are designed for the shared nothing hardware paradigm. This includes Informix XPS [Ger95], IBM DB2/PE [BFG+95], Sybase Navigation Server [Syb], Tandem [Tan87], and Teradata [Ter83]. While one can certainly run a shared nothing system on an SMP (using the shared memory as a fast communication network), SMP's offer alternatives to algorithm design not present in shared nothing machines which include load balancing and easier management (as there are fewer independent nodes). The question is: should we redesign all the query evaluation algorithms specifically for the SMPs or will the shared nothing algorithms suffice? In other words, do we need to maintain two software architectures, one for SMPs and the other for shared nothing hardware? The answer is not clear. On one hand, SMPs allow simple algorithms due to shared memory; they require none of the massive data redistribution that is inherent in shared nothing parallel join algorithms. This suggests that shared memory algorithms should dominate shared nothing algorithms on SMPs. On the other hand, naive shared memory algorithms have poor cache behavior due to their lack of locality of memory reference. Shared-nothing algorithms have better locality of reference, which suggests that they should be faster than shared memory algorithms. Hence it is not clear how big the difference in performance between shared memory and shared nothing algorithms on SMPs will be; it is not even clear which class of algorithm will dominate.

To begin to answer the question, we studied one part of parallel DBMS performance: the memory resident portion of join evaluation on an SMP, with an implementation on the SGI Challenge multiprocessor. Since the goal of our study is to compare options for architecting the memory resident portion of the join processing, by not including I/O costs in our performance study, we are perhaps exaggerating the differences between the algorithms. That is, if we added I/O to all of the algorithms, the running time of each would shift upwards by the same amount, making the deltas between the running times a smaller fraction of the total running time. The main conclusion of our paper, detailed below, is that optimized shared nothing algorithms and shared memory algorithms have similar performance on SMPs; this conclusion would only be supported more strongly if we added in I/O. To put it another way, our study is biased to emphasize differences between algorithms. It was surprising to us to find that even given this bias the algorithms did not differ significantly except where memory copy overheads dominated.

We first studied the performance of the shared nothing algorithms on an SMP. First we implemented a transparent shared nothing-like message passing library. We found that the algorithm performs quite poorly using such a library. Then we optimized the message passing by explicitly exploiting the shared memory so as to avoid any extra memory copies which resulted in a restriction that a sent buffer could not be reused. However, the performance of the algorithm was 90–190% better as there were no extra memory copies.

Then we studied the performance of the traditional shared memory hash join algorithm which extends from the uniprocessor hash join where all nodes build a shared hash table and then each node probes its tuples in the shared hash table [LTS90]. The traditional algorithm

and its optimized version performed only marginally better than the shared nothing algorithm with optimized message passing. Next we attempted to optimize the shared memory algorithm in terms of better memory contention and locality by using repartitioning in shared memory, instead of using messages. However, these algorithms performed only comparably to the traditional simple approach. Finally we exploited the fact that the repartitioning algorithm can be cache optimized as the join processing after repartitioning is local to a processor [SKN94]. The cache optimized algorithms performed 9–90% faster than the traditional shared memory algorithm.

In related work, [LTS90] did an analytical performance of algorithms in a shared everything environment. However, they considered different basic algorithms comparing hybrid hash, hash loop, etc. The analytical model did not truly reflect an SMP environment. Furthermore, the considered data domain was too restricted which resulted in misleading conclusions. [SKN94] shows that the cache misses on a uniprocessor machine are expensive as the data has to be fetched from (slow) main memory and therefore the algorithms must be designed taking the cache into account. An access to a shared variable is even more likely to result in a cache miss because of the maintenance of cache coherence. Consequently, cost of data sharing can not be ignored while designing efficient query processing algorithms for the SMPs. [CLR94] shows that for some scientific applications, optimized shared memory and message passing programs perform comparably on comparable hardware.

The remainder of the paper is organized as follows. In Section 2 we review the basic SMP architecture. Section 3 describes the experimental platform. We study the traditional approaches, the hybrid and the cache-conscious algorithms for the SMPs in Section 4 followed by their comparative performance evaluation in Section 5. Conclusions are offered in Section 6.

2 Review of SMP Architecture

The generic structure of a shared memory multiprocessor is as follows. There is a shared address space and each processor has a private cache memory for speeding up the access to the shared address space. Thus the processors rely on the cache memory for achieving high performance as going to the main memory is fairly expensive. Commonly, in the SMP architecture the processors share the memory bus and the main memory as illustrated in Figure 1.

The data in the private cache memory is kept coherent. Coherence implies that the data values are guaranteed to be consistent under some coherence semantics, the most common being the sequential semantics i.e. a read returns the value of the most recent preceding write. There are several cache coherence algorithms mentioned in the literature [AB86]. For SMPs, which have a shared bus, a common algorithm is the snooping bus write invalidate. In this algorithm, all cache controllers listen on the bus and when a write occurs on a cached address, the cached value is invalidated. This is possible because the cache is write through i.e. all writes go to the memory. On a read, the value is read into the cache from the shared memory.

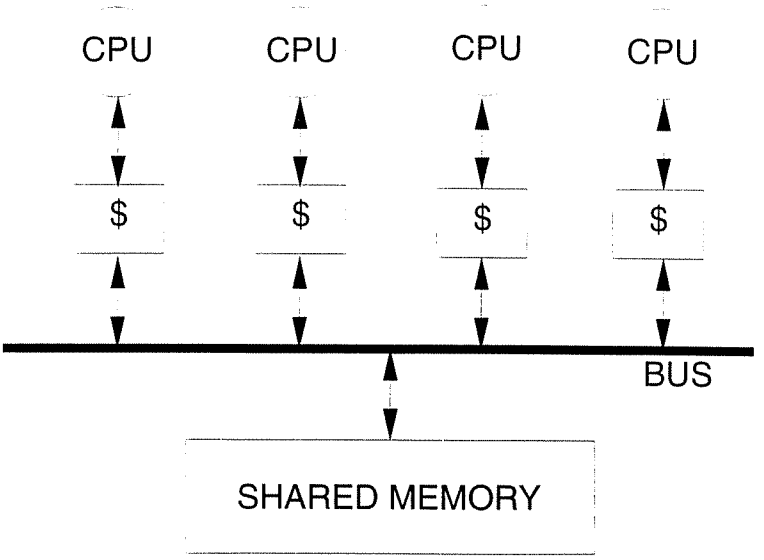

Figure 1: Block Architecture of an SMP

For example, a "message," i.e. a write to an address followed by a read of the same address on a different processor, necessarily requires two (slow) memory accesses under the write invalidate protocol, one for write and another for read as any cached copy will be invalidated on the write. Thus making the communication of the data value comparatively slow compared to non-shared memory accesses most of which are expected to be hits in the local cache.

The naive PRAM (Parallel Random Access Memory) model of parallel computation assumes that all memory accesses are of uniform cost. This assumption is commonly made in design and analysis of query processing algorithms. However, it is clear that SMPs do not match the PRAM model. In fact, another, and perhaps better, way to look at the shared memory machine is to assume that the machine is a distributed-cache machine and that the communication takes place when two processors access the same address at different points of time.

The main question that remains is whether the communication cost is sufficiently large so as to justify changing our programming model implying designing the algorithms to minimize communication. In the remainder of the paper we show that 1. doing pure message passing (shared nothing) is viable, and 2. PRAM model is reasonable but designing algorithms taking the SMP architecture—shared memory, caches and not-for-free communication—performs better.

3 The Experimental Platform

We compare the performance of the algorithms by implementing them on a SGI Challenge SMP. The hardware configuration of the SMP is 12 MIPS R4400 processors, 1 GB of shared address space and 1 MB off-chip private cache memory per processor. There is a 16KB data cache

on-chip. The performance of the algorithms is compared on five different work loads detailed in table 1 where both relations are equally large.

#	work load	approx. result size
1	a tuple of R matches many ($\approx$8) tuples of S	$8 * \|R\|$
2	a tuple of R matches zero (with $\frac{7}{8}$ prob.) or many ($\approx$8) tuples of S	$\|R\|$
3	same as 2 except R and S are changed	$\|R\|$
4	a tuple of R matches one tuple of S on average	$\|R\|$
5	a tuple of R finds a match in S with ($\frac{1}{8}$) prob.	$\frac{1}{8} * \|R\|$

Table 1: Description of Experimental Work Loads

The relative performance of the algorithms is sensitive to the join selectivity, which affects how repeated memory access intensive the join is. The work loads are designed to explore the range of join selectivity from large to small as described below where relation R is used for building the hash table, and relation S is probed in the hash table. Other workloads studied did not give any further insight.

wl#1 Since a tuple of S matches many tuples of R, the hash table is accessed multiple times. The work load represents the work loads with high join selectivity where a tuple matches several probe tuples and the result size is larger than the input relation.

wl#2,3 These work load represent the case of key, foreign key join, with a possible select on the foreign key. A tuple can match either zero or several tuples of the other relation.

wl#4 This is an ideal case where each tuple matches about one tuple of the other relation resulting a result size similar to that of input relations.

wl#5 In this work load a tuple of one relation matches zero or one tuple of the other relation, possibly due to a select, resulting in a small join selectivity.

The timing runs were obtained by running the algorithm multiple times and taking the average after ignoring the best and worst runs. We used 8 processors with 40K tuples (104 byte/tuple) per node. The join attribute was 16 bytes long. We also ran scaleup experiments using work load # 4 above, varying the number of processors from 2 to 8.

4 Parallel Hash Join Algorithms

The basis of all parallel hash join algorithms is the uniprocessor hash join. The basic uniprocessor in-memory hash join algorithm works as follows. Assume R and S are the two relations (or fragments of relations of a bigger join) being joined. Furthermore, the relations are now in the main memory in slotted pages after having been read from the disk or are in the memory

of a main memory DBMS. In the hash join algorithm, first a hash table of the tuples of R is created by hashing them on the join attribute. Then the tuples of the relation S are probed in the hash table by hashing them on the join attribute and searching the attribute value in the hash table. The matching tuples are output.

First we studied the performance of the shared nothing approach on the SMPs to evaluate its viability.

4.1 The Shared Nothing Approach

Shared nothing algorithms are generally based on the assumption that the communication between processors is slow. A common, though not necessary, feature of a shared nothing hardware is that the communication in the algorithms is through message passing. Two factors influence the performance of any message passing library: 1. the underlying communication hardware and 2. how well the library is implemented. Though it is common to ignore the second effect, it is nevertheless important and much work has gone into making it more efficient.

As evident from the SMP architecture discussed earlier, the hardware cost of communication in an SMP is roughly equivalent to doing main memory accesses, instead of cache accesses, accompanied with some cache coherence protocol overheads. This is generally much faster than speeds achievable in a traditional MPP interconnection network. Since the hardware is fairly fast, the software cost of message passing becomes important and we find the same in our experiments detailed below.

A shared nothing hash join algorithm, as in Gamma [DGS+90], first repartitions the relations by hashing the tuples on the join attribute, each hash value being assigned a specific node. Each node then joins the partitions of the relations locally as if it were the only node in the system. A property of the algorithm is that a tuple is moved only once across the nodes minimizing communication between nodes. Tuples are actually sent in large (4 KB in our case) batches, except the last one, to minimize the communication cost. The algorithm, in two threads running in parallel, is as follows.

```
/* node i, repartitioning thread */
foreach t in R_i
  hash t to find the destination node p
  send t to node p /* in 4KB batches */

foreach t in S_i
  hash t to find the destination node p
  send t to node p /* in 4KB batches */


/* node i, join thread */
while (t in R not exhausted)
  receive t of R sent to me
  insert t in the hash table
```

```
while (t in S not exhausted)
  receive t of S sent to me
  probe t in the hash table
  if (match)
    generate output tuple
```

In shared nothing architectures, the repartitioning involves sending the tuple over an interconnection network across the node boundaries using a message passing library. The question as mentioned earlier is: how will the algorithm perform if the SMP provides a message passing library just like a shared nothing architecture?

Since no standard implementation of message passing, like MPI [Mes94], was available on the machine, we implemented two different message passing libraries to evaluate the performance of the shared nothing algorithm. In the first, sending and receiving a message involves the following steps:

1. copying the source buffer provided by send() to a library buffer which is augmented with some header information like sender id,

2. letting the receiving process know that it has a pending message by putting the message in a queue for that process,

3. upon a receive() copying the library buffer to the buffer supplied by the receive() function and returning some of the relevant header information like length of the message.

The library is quite transparent to the rest of the algorithm and is therefore similar to a message passing library on a shared nothing system. However, it has two memory copy operations for each send/receive pair. The shared nothing algorithm using the transparent library is henceforth called the Transparent Shared Nothing algorithm.

In the second version we exploited the shared memory and eliminated all extra memory copies. We call this version the Optimized Shared Nothing algorithm. In this version, the sending and receiving of a message involves the following steps:

1. the buffer supplied by the send() can't be reused by the sending process and is augmented with some header information like sender id,

2. the receiving process is notified of a pending message by placing the message in a queue for the process,

3. upon a receive(), a pointer to the buffer and some relevant information, like length of the message, is returned to the receiving process and the receiving process can use the buffer as its own.

As is evident, there are no memory copies involved in the sending of a message. However, this approach is not transparent because the sending process can not reuse a sent buffer as

buffer address now "belongs" to the receiving process. This, of course, is not the case in a real message passing system where the address spaces are completely disjoint, but should reflect how efficient can we be if we want to directly use the shared nothing algorithms on the SMPs in order to reduce software development cost.

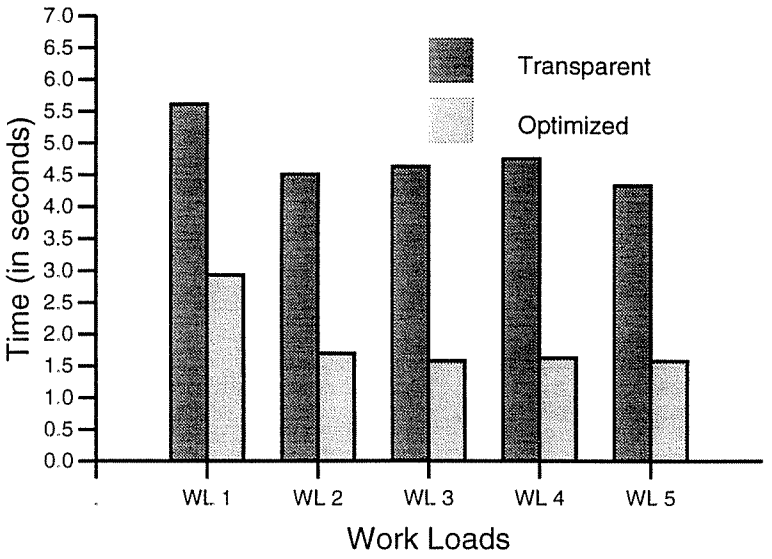

Figure 2: Performance of the Shared Nothing Approach

The performance of the shared nothing algorithm with the two libraries is shown in Figure 2. It shows that, as expected, the Optimized Shared Nothing algorithm performs significantly better showing a performance gain of 90–190% because it does fewer memory copy operations. As shown later, the Optimized Shared Nothing algorithm compares well with the other shared memory algorithm showing that the shared nothing approach is viable on an SMP.

In order to investigate the relative performance of the shared nothing algorithm further, we evaluated the performance of the simple shared memory approach described below.

4.2 The Shared Memory Approach

The uniprocessor hash join algorithm is easily extended for SMPs as follows. All processors read their partition of the relation R and build the (global) hash table by hashing the tuples on the join attribute. The access conflicts to the hash table are taken care of by latching. Thereafter, all processors read their partition of the relation S tuples and probe them in the shared hash table. This approach is explicitly used in [LTS90] and is implicit in the XPRS database system [HS91].

The Tuple-based Shared Memory algorithm, detailed below, builds the hash table on the tuples themselves. Under the constant time memory access (PRAM) model this algorithm is the most efficient as it does the least amount of "work" i.e. it has least number of memory accesses (and instructions). Hence we use it as a basis for comparison with all other algorithms.

```
/* node i */
foreach t in R_i
  hash t to find the bucket b of the hash table
  latch bucket b to avoid conflicts
    insert t in the hash table bucket b
  unlatch bucket b

foreach t in S_i
  hash t to find the bucket b of the hash table
  probe t in the hash table bucket b
  if (match)
    generate output tuple
```

At the outset, this algorithm looks ideally parallel, except for the small latching overhead to handle access conflicts to the hash table in the build phase. However, a look inside the functioning of the algorithm on an SMP shows a few shortcomings.

1. Latching overhead may be non-trivial as latching can be expensive in SMPs.
2. Access to the shared hash table has extremely poor processor locality of access because any processor at random may access a bucket of the hash table.
3. A very small portion of the hash table can fit in the cache, effectively reducing the speed at which the hash table may be accessed.

A possible variation and optimization to the algorithm is to extract the join attribute and keep a pointer to the original tuple, as is done in sorting, before building or probing the tuples. This should improve the effective utilization of the cache as the join processing doesn't need to access the entire tuples except when building the result tuples but it incurs the small overhead of the extraction of the join attribute from the tuple. We call this the Attribute-extraction Shared Memory algorithm.

```
/* node i */
foreach t in R_i
  hash t to find the bucket b of the hash table
  extract attribute/pointer from t into e_t
  latch bucket b to avoid conflicts
    insert e_t in the hash table bucket b
  unlatch bucket b

foreach t in S_i
  hash t to find the bucket b of the hash table
  extract attribute/pointer from t into e_t
  probe e_t in the hash table bucket b
  if (match)
    generate output tuple
```

As mentioned earlier, the Tuple-based Shared Memory algorithm is expected to perform the best. However, the performance study shows that the attribute/pointer extraction does improve the performance of the basic algorithm in most cases as evident in Figure 3. In the first work load, the result size is larger than the input relation size i.e. a *build* tuple is being accessed multiple times. Hence the attribute extraction is not of much use because one has to access the tuple in the buffer page multiple times anyway eclipsing the optimization. On the other hand, when the tuple is accessed only once (or less) then attribute extraction pays off because many comparisons do not involve fetching the tuple. One may note that the relative performance gain of the Attribute-extraction Shared Memory algorithm is even larger for the last work load as expected because it has a small result size.

Finally, we notice that the Optimized Shared Nothing algorithm performs only a little worse (in fact better for work load 1) than the Tuple-based Shared Memory algorithm thus showing that it is a viable alternative on an SMP given that the message passing is highly tuned.

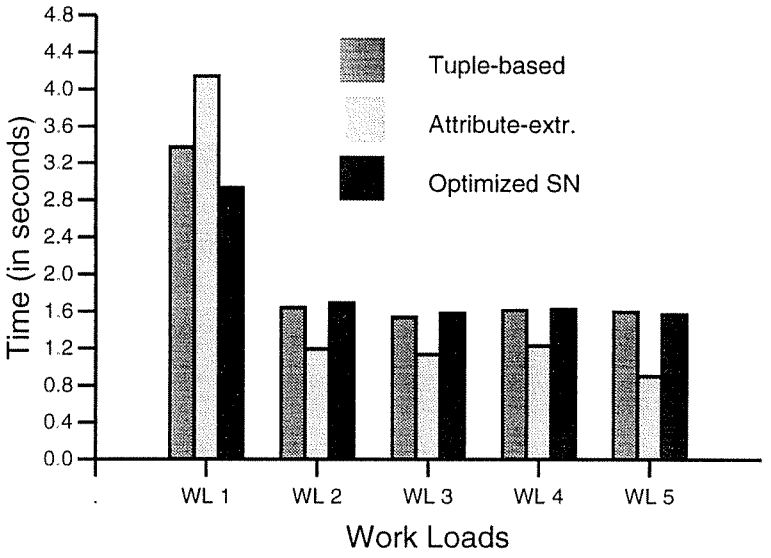

Figure 3: Performance of the Shared Memory Approach

However, the traditional approach is not the only way of performing a hash join on an SMP. In order to achieve superior performance, we attempted eliminating the shortcomings of the shared memory approach. The first two shortcomings which are present in the shared memory technique can be removed by borrowing some characteristics of the shared nothing approach. This mainly includes repartitioning the relations using shared memory instead of sending messages before performing the local join.

4.3 The Hybrid Approach

The hybrid approach adapts the shared nothing paradigm of repartitioning and then performing the local join on SMPs. The algorithms first repartition the relations by hashing on the join

attribute. Then each processor performs a local join independently as in a shared nothing algorithm as detailed in pseudo-code below.

```
   /* node i */
   foreach t in R_i
     hash t to find the destination node p
1    repartition t by putting it in a "run" for node p

   foreach t in S_i
     hash t to find the destination node p
2    repartition t by putting it in a "run" for node p

   foreach t of R from each processor "run" for node i
     insert t in the hash table

   foreach t of S from each processor "run" for node i
     probe t in the hash table
     if (match)
       generate output tuple
```

Whereas in shared nothing machines, the repartitioning involves sending the tuple itself to the buffer pool of the destination node, the shared memory permits different ways of repartitioning the relations. The details of the repartitioning, lines 1 and 2 above, differentiate the three algorithms.

1. Repartitioning the relation by copying the tuples to the local buffer pool of the destination node, henceforth called the Tuple-copy Hybrid algorithm. This is similar to doing efficient message passing in terms of memory copy costs.

2. Leaving the tuples in the original (shared) buffer pool and creating partitions on the destination node by having pointers to the tuples in their original buffer pool, henceforth called the Pointer-based Hybrid algorithm.

3. Like the approach 2 above, but the partitions also have a local copy of the join attribute along with the pointer to the tuple, henceforth called the Attribute-extraction Hybrid algorithm. (This is similar to the attribute/tuple pointer extraction optimization because it improves the spatial locality of access for the join attribute.)

The performance of the three variants in Figure 4 shows the following.

1. Analogous to the shared memory case, the Attribute-extraction Hybrid algorithm performs best when the result sizes are small. When the result size is large i.e. a tuple is being accessed multiple times, as in work load 1, the attribute extraction is not of much use because one has to access the tuple in the buffer page multiple times anyway to build the result tuple making the algorithm slower.

2. Though Pointer-based Hybrid algorithm has less partitioning overhead, it performs quite poorly because one has to go through the pointer to access the join attribute and also to access the tuple, both possibly off the local cache.

3. In fact, the Tuple-copy Hybrid algorithm performs better than the Pointer-based Hybrid algorithm showing that the overhead of tuple copying is significantly compensated for by the increase in the locality of access.

4. Overall, we notice that the overhead of repartitioning is large and it eclipses the gain in the performance of the join due to removal of latching and improvement in processor locality. Hence the algorithms perform only comparably to the Tuple-based Shared Memory algorithm.

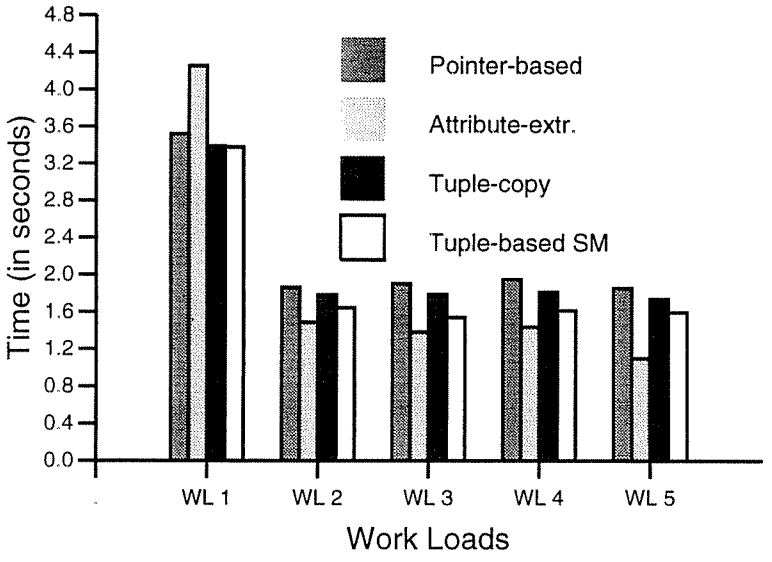

Figure 4: Performance of the Hybrid Approach

The main reason for the poor performance is that the algorithms still has very poor cache behavior though it has incurred the overhead of repartitioning. This is because even though the tuples are partitioned, the partitions are large enough that only a small portion of the hash table can fit in the cache, effectively reducing the speed at which the hash table may be accessed. Therefore, though the misses occurring in cache due to cache coherence are reduced by the repartitioning, the algorithm still suffers. Partitioning the data further into small cache-size partitions is likely to mitigate this shortcoming, as now the entire partitioned hash table is likely to fit in the cache while being accessed [SKN94]. Thus, the hash table is likely to be entirely cache resident while being accessed. Note that the this optimization does not add any noticeable additional overhead over the hybrid approach but should improve the performance of the algorithm resulting in the following cache conscious algorithm.

4.4 Cache Conscious Hash Join

In this algorithm, instead of just hash repartitioning the relations on the join attribute across nodes, we further divide each of the partitions into cache size units by hashing on the join attribute. Each processor then repeatedly processes each of the corresponding cache size units of R and S. The partitioning of the relations is achieved by letting each processor partition its tuples and write them in a "run" for each cache size work unit. After the partitioning phase is over, a processor will have to read all the runs belonging to one cache size unit.

```
/* node i */
foreach t in R_i
  hash t to find the node p, and cache-size partition r
  repartition t by putting it in a "run" for node p, partition r

foreach t in S_i
  hash t to find the node p, and bucket b of the hash table
  repartition t by putting it in a "run" for node p, partition r

foreach cache-size partition r of my node
  foreach t of R from each processor "run" for node i, partition r
    insert t in the hash table
  foreach t of S from each processor "run" for node i, partition r
    probe t in the hash table
    if (match)
      generate output tuple
```

While partitioning the relation we extract the join attribute and only keep a pointer to the tuple because as shown previously this has the best performance in most cases for the hybrid approach.

We expect this algorithm, called the Partitioned Cache Conscious algorithm, to do better because first, it minimizes data sharing across the processors as partitioned tuples never need to be accessed by two processors and second, the partitioned hash tables are likely to be cache resident thus making the access to the hash table effectively much faster. If the overhead of partitioning is sufficiently small, the algorithm is expected to outperform the other algorithms.

In a variation of the algorithm, called the Queue-based Cache Conscious algorithm, detailed below, a partition of R and corresponding partition of S are considered as a work unit. Instead of being preallocated to processors, as above, these partitions are placed in a work queue. Each processor, when idle, picks up the next unit from the queue and processes it. The processing stops when the work queue gets empty. The work queue approach is possible due to the shared memory. This algorithm could outperform the first one if there is a small non-uniformity in the distribution of work load or if the system does not make the processors run as evenly as is theoretically possible because of OS functions or other system/user processes. In such cases the inherent load balancing of the algorithm, despite the small overhead of accessing the queue, is likely to result in superior performance.

```
/* node i */
foreach t in R_i
  hash t to find the cache-size partition r
  repartition t by putting it in a "run" for cache partition r

foreach t in S_i
  hash t to find the node p, and bucket b of the hash table
  repartition t by putting it in a "run" for cache partition r

while there is a cache-size partition r remaining in the queue
  foreach t of R from each processor "run" for cache partition r
    insert t in the hash table
  foreach t of S from each processor "run" for cache partition r
    probe t in the hash table
    if (match)
      generate output tuple
```

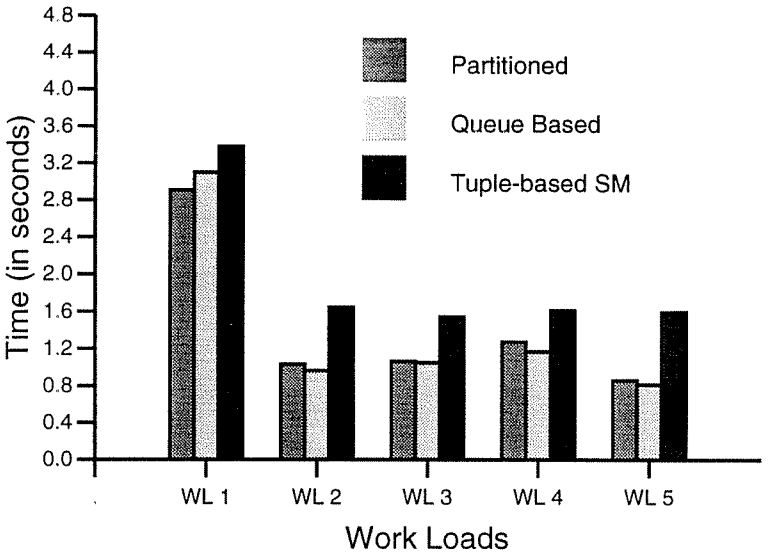

Figure 5: Performance of the Cache Conscious Algorithms

Figure 5 compares the performance of the two cache conscious approaches and the Tuple-based Shared Memory algorithm. It shows that the Queue-based Cache Conscious algorithm usually outperforms the Partitioned Cache Conscious algorithm despite the additional overhead of maintaining and accessing the work units from a queue which shows that the small variations in the different nodes can be successfully overcome using the queue based approach. Furthermore it shows that doing cache optimization using repartitioning, cache-size partitioning, and attribute pointer extraction does have significant, 9–90%, performance improvement over the basic Tuple-based Shared Memory algorithm.

5 Performance Summary

The above discussion is summarized in Figure 6. It compares the performance of the Transparent and Optimized Shared Nothing algorithms, the Tuple-based shared memory algorithm and the Queue-based Cache Conscious algorithm.

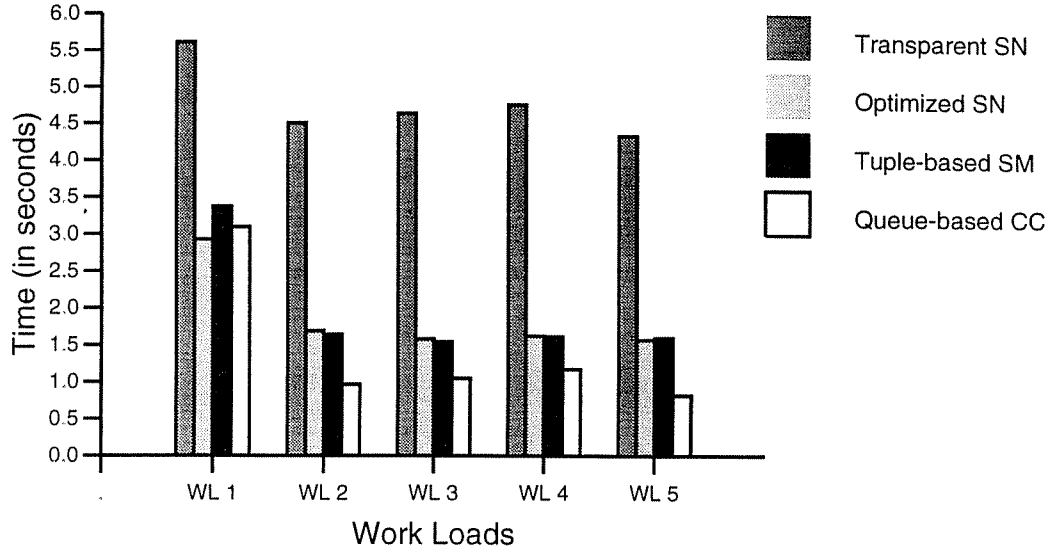

Figure 6: Relative Performance of the Algorithms

From the figure it is evident that:

1. The Shared Nothing approach is viable on an SMP but it needs to use an optimized message passing library that minimizes memory copies.

2. The Tuple-base Shared Memory algorithm performs marginally better for most workloads. However, for the first workload which is memory access intensive as it has a high join selectivity, the shared nothing algorithm performs better because it has better memory access contention and locality.

3. Specializing algorithms for the SMPs architecture, exemplified by the Queue-based Cache Conscious algorithm, does have performance dividends.

Figure 7 shows the scaleup characteristics of the Queue-based Cache Conscious, the Optimized Shared Nothing and the Tuple-based Shared Memory algorithms on work load 4. It shows that all three algorithms have comparable scaleup performance.

6 Conclusions

Our study shows that running traditional shared nothing algorithms on SMPs performs acceptably well provided that the message passing library is optimized to remove unnecessary

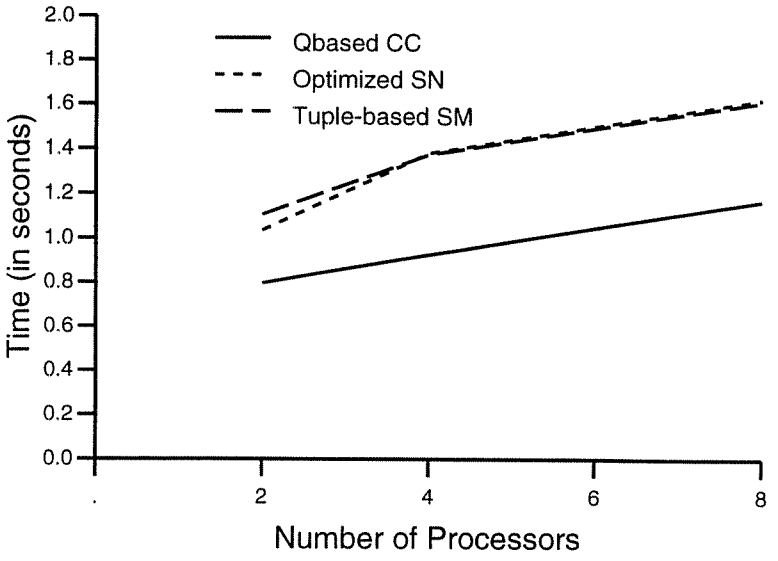

Figure 7: Scaleup Characteristics of the Algorithms

memory copies. The shared memory algorithms perform marginally better than the shared nothing algorithm. The performance of the shared memory algorithms can be enhanced by optimizing the algorithms by making them aware of the cache and the SMP architecture.

One argument against shared nothing algorithm has been their poor performance when the data is skewed [WDJ91]. However, techniques that have proven effective for shared nothing algorithms, e.g. [SN93], would trivially apply to SMPs. It would be interesting to compare the two approaches for skew handling. Also, cluster of SMPs seems to be gaining popularity for building large scalable database systems, instead of the traditional shared nothing hardware. Investigating the algorithms for a cluster of SMPs is part of our future work.

References

[AB86] J. Archibald and J.-L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, 1986.

[BFG+95] Chaitanya Baru, Gilles Fecteau, Ambuj Goyal, Hui i Hsiao, Anant Jhingran, Sriram Padmanabhan, and Walter Wilson. An Overview of DB2 Parallel Edition. In *Proc. of the 1995 ACM-SIGMOD Conference*, San Jose, CA, 1995.

[CLR94] Satish Chandra, James R. Larus, and Anne Rogers. Where is time spent in message-passing and shared-memory programs? In *Proc. of the 6th ASPLOS Conference*, pages 61–75, October 1994.

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

[Ger95] Bob Gerber. Informix Online XPS. In *Proc. of the 1995 ACM-SIGMOD Conference*, San Jose, CA, 1995.

[HS91] W. Hong and M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. In *Proceedings of the 1st Int'l Conf. on Parallel and Distributed Information Systems*, Miami, Florida, December 1991.

[LTS90] Hongjun Lu, Kian-Lee Tan, and Ming-Chien Shan. Hash-Based Join Algorithms for Multiprocessor Computers with Shared Memory. In *Proceedings of the 16th VLDB Conference*, pages 198–209, Brisbane, Australia, September 1990.

[Mes94] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. *Int'l Journal of Supercomputing Applications*, 8(3/4), 1994.

[SKN94] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proc. of 20th Int'l Conference on Very Large Data Bases*, pages 510–521, Santiago, Chile, September 1994.

[SN93] Ambuj Shatdal and Jeffrey F. Naughton. Using Shared Virtual Memory for Parallel Join Processing. In *Proc. of the 1993 ACM-SIGMOD Conference*, pages 119–128, Washington, D.C., May 1993.

[Syb] Sybase Inc. Sybase Navigation Server
URL: http://www.sybase.com/Offerings/Servers/navserver.html.

[Tan87] Tandem Database Group. Nonstop SQL, A Distributed, High-Performance, High-Reliability Implementation of SQL. In *Workshop on High Performance Transaction Systems*, Asilomar, CA, 1987.

[Ter83] Teradata Corp. *Teradata: DBC/1012 Database Computer Concept and Facilities*, 1983.

[WDJ91] C. B. Walton, A. G. Dale, and R. M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proceedings of the 17th VLDB Conference*, pages 537–548, Barcelona, Spain, September 1991.