Parallel Pointer-Based Join Algorithms in Memory Mapped Environments

Peter A. Buhr, Anil K. Goel, Naomi Nishimura, Prabhakar Ragde
Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1
{pabuhr,akgoel,nishi,plragde}@uwaterloo.ca

Abstract

Three pointer-based parallel join algorithms are presented and analyzed for environments in which secondary storage is made transparent to the programmer through memory mapping. Buhr, Goel, and Wai [11] have shown that data structures such as B-Trees, R-Trees and graph data structures can be implemented as efficiently and effectively in this environment as in a traditional environment using explicit I/O. Here we show how higher-order algorithms, in particular parallel join algorithms, behave in a memory mapped environment. A quantitative analytical model has been developed to conduct performance analysis of the parallel join algorithms. The model has been validated by experiments.

1 Introduction

Programmers working with complex and possibly large persistent data structures are faced with the problem that there are two, mostly incompatible, views of structured data, namely data in primary and secondary storage. In primary storage, pointers are used to construct complex relationships among data; establishing these relationships without pointers is often cumbersome and expensive.

Significant research has occurred over the last decade, starting with the seminal work by Atkinson and Morrison [8, 9], on efficient and simple-to-use methodologies for constructing, storing, and subsequently retrieving large persistent data structures in a fashion that makes the secondary storage transparent to the programmer. The approaches extend primary storage practices and tools so that they also apply to secondary storage. Merging primary and secondary storage in this way produces a *single-level store*, which gives the illusion that data on secondary storage is accessible in the same way as data in primary storage. This uniform view of data eliminates the need for expensive execution-time conversions of structured data between primary and secondary storage and allows the expressive power and the data structuring capabilities of a general purpose programming language to be used for secondary storage. Although a single-level store was investigated as far back as the Multics system (1968) [26], it has seen only limited use even in the field of operating systems; only in the last few years has this approach blossomed in both the database and programming language communities [28, 13, 32, 21]. For complex data structures, a single-level store offers substantial performance advantages over conventional file access, which is crucial to database applications such as computer-aided design, text management, and geographical information systems.

While there are several ways to implement a single-level store, many projects do so using memory-mapping. Memory mapping is the use of virtual memory to map files stored on secondary storage into primary storage so that the data is directly accessible by the processor's instructions. In this environment, there are no explicit read and write routine calls to access data on disk. All read and write operations are done implicitly by the operating system during execution of a program. When the working set of an application can be kept entirely in memory, performance begins to approach that of memory-resident databases.

The goal of this work is to investigate the behaviour of existing database algorithms in a particular kind of memory mapped environment, especially in highly parallel systems. The algorithm chosen as the starting point for this work is the join algorithm: "Because any data model supporting sets and lists requires at least intersection, union, and difference operations for large sets, we believe that [the discussion of join algorithms] is relevant to relational, extensible, and object-oriented database systems alike." [17] In particular, this work examines parallel versions of nested loops, sort-merge, and Grace [20] algorithms for memory mapped environments. A quantitative analytical model has been designed and validated through experiments for each parallel join algorithm. Our hope is that the model will act as a high-level filter for data structure and algorithm designers to predict general performance behaviour without having to construct and test specific approaches. More importantly, a quantitative model is an essential tool for subsystems such as a query optimizer.

2 Related Work

The influences on our work stretch across a number of areas within computer science. We divide our brief survey of the literature into three areas: persistence through memory mapping, theoretical I/O modelling, and other studies on database joins in shared-memory environments.

2.1 Related Memory-Mapping Approaches

All approaches to implementing memory mapped single-level stores have to deal with the *address consistency* problem. When data is copied from secondary to primary storage, either the data must be positioned exactly where it was created (to maintain integrity of references), or the addresses must be modified to reflect its new location. The former case is difficult to handle because data from two or more memory mapped files may map to the same location, producing an irreconcilable conflict. The latter case is difficult to handle because it must be possible to locate all pointers so they can be updated, and there is the additional runtime cost of modifying the pointers. Pointer modification may be handled eagerly or lazily; in general, eager modification of pointers is called *relocation* and lazy modification is called *pointer swizzling* [12, 25].

Objectstore [21] is a commercial product that uses pointer relocation, Paul Wilson has developed related

12th International Conf. on Data Engineering, Feb 1996

pointer swizzling schemes [38], and other pointer relocation schemes are appearing, such as QuickStore [37]. However, we argue that a significant performance advantage of a single-level store is lost if all the pointers within it have to be relocated or swizzled. This loss of advantage is especially significant for operations that incur high overhead in data preparation; examples include operations like sequential scans, where the data is accessed only once, and operations that deal with large data structures with small primary storage, where the data is implicitly fetched and prepared multiple times. Therefore, we are pursuing the alternative approach of exact positioning of data so relocation or swizzling of pointers is significantly or completely eliminated during transfers to and from secondary storage.

To this end, we are developing μDatabase [11], a toolkit for building persistent data structures using the "exact positioning of data" approach to memory mapping. μDatabase employs a novel technique that allows application of an old solution to the problem of address collisions when mapping multiple memory mapped data structures. The old solution is hardware segmentation; each hardware segment is an address space starting at a virtual zero, in which a persistent data structure can be built, stored, and subsequently retrieved and modified. Multiple segments can be simultaneously accessed in a single application because each segment has its own non-conflicting address-space. When a segment is mapped into memory, pointers *within* the segment do not require modification; pointers outside the segment do require modification, but in general, these pointers represent a small percentage of the total number of pointers in a data structure. Our technique uses the UNIX system call `mmap` to mimic segmentation on hardware that does not support it. Furthermore, μDatabase has direct access to the concurrency facilities provided by μC++ [10], which allows a high level of concurrency through multi-threading.

The main disadvantage of our technique is the need to perform additional copying when transferring data from segment to segment. When the hardware does not support segmentation, no inter-segment copy instruction exists. Therefore, it is necessary for segments to share some portion of their address space for transferring information; hence, our segments have an address space that is divided into private and shared portions (see [11] for details).

2.2 Related Theoretical Models

In recent years, attempts have been made to model the I/O bottleneck, and spatial and temporal locality from within the theoretical framework. These models build on the framework of the sequential RAM [5] and its parallel variant, the PRAM [14]. The first step towards a more realistic memory model is distinguishing between local and global memory [27, 2], yielding a two-level memory scheme. More recently, there have been attempts to model multi-level memory [1, 3, 6], both in sequential and parallel settings. The notions of block transfer and hierarchy are developed further in a parallel model in which memory consists of a tree of modules, where computation takes place at the leaves [6]. I/O complexity models start with a single disk and CPU with block transfer [18, 4] and continue through parallel disks with flat memory and hierarchical memory [35, 36]. Our analytical model draws on ideas from several of these papers, though our intent is not to characterize the complexity of problems, but rather to predict performance on many real architectures.

2.3 Related Database Studies

Our work builds on the framework proposed by Shekita and Carey [33]. They present an unvalidated single-processor, single-disk model for three pointer-based join algorithms: nested loops, sort-merge, hybrid hash. Our model is multiprocessor, multi-disk and removes or modifies a number of simplifying assumptions made by Shekita and Carey. They assume the cost of I/O on a single byte to be a constant, not taking into account seek times or the possibility of savings using block transfer; they do not distinguish between sequential and random I/O; they do not consider the fact that the minimum I/O transfer unit on virtually all computers is at least a disk sector and more commonly a page. For the hybrid-hash algorithm, two assumptions made in their paper need to be extracted from the analysis: constant-time hashing, and clustering of identical references in a single hash chain. We replaced the second assumption with the weaker assumption that all objects of the inner relation referenced in one hash chain can fit into the portion of memory not used by the hash table.

Shapiro [31] analyzes sort-merge and three hash-based algorithms and also provides a discussion of various memory-management strategies. Again, no experimental data is provided to validate the model.

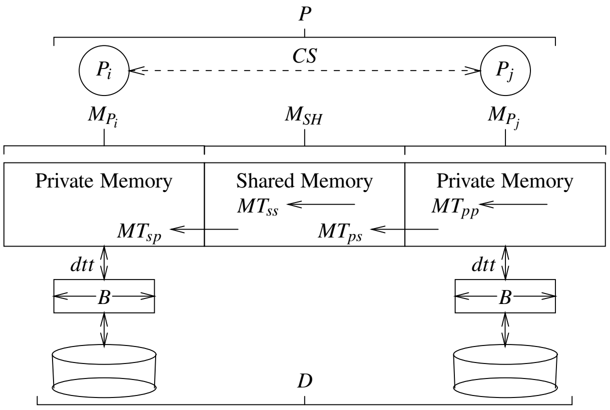
Lieuwen, DeWitt and Mehta [22] analyze parallel versions of Hash-Loops and Hybrid-Hash pointer-based join algorithms and compare them to a new algorithm, the Probe-child join algorihtm. Their work also builds upon [33] but has a different emphasis from our work in that we develop a validated model for a shared memory architecture based upon the memory mapping approach.

Martin, Larson and Deshpande [24] present a validated analytical model for a multi-processor, single disk situation in a shared-memory based traditional system. Their model assumes perfect inter-process parallelism and perfect processing-I/O parallelism. We make neither assumption in our model.

Our work extends the work in the above papers in a number of ways: by allowing multiple processors and multiple disks (resulting in further algorithm design decisions in the course of parallelizing the standard join algorithms mentioned here), by drawing a distinction between private and shared memory, and of course by using a memory mapped environment. The parallelization used in our algorithms has been influenced by ideas presented in [30]. In addition, our analysis is quantitative as opposed to the qualitative analysis in other models; our model has measured parameters that quantify the environment in which the join occurs, such as how disk I/O is affected by all aspects of the join.

3 Modelling

Our model has as components a number of processes, each having its own segment with a private area of memory and a shared area of memory accessible to all processors through which communications takes place, and a number of disks allowing parallel I/O. The parameters of the model are discussed below and shown graphically as:



The number of processes used by a given algorithm is P. The time for a context switch between processes is CS. The number of bytes of private memory used by a process P_i is M_{P_i}. The number of bytes of shared memory available for use by the P processes is M_{SH}. The size of a block or page of virtual memory, in bytes, is B.

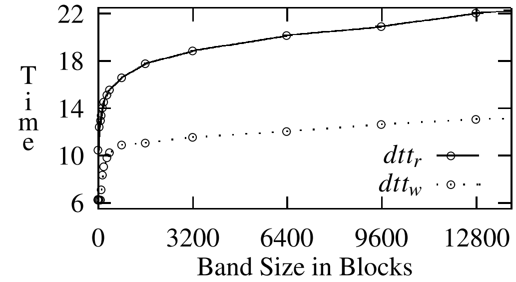
The number of parallel I/O operations is D. This parameter usually refers to the number of disk controllers, not disks, since there is a one-to-many relationship between controllers and disks. When simultaneous requests arrive for the same disk, we leave unspecified the disk arbitration mechanism. Alternatives include denying algorithms simultaneous access, serializing overlapping requests, and a priority scheme for simultaneous requests.

Memory transfer times are given in the form of combined read/write times since all segment transfers move data using assignment statements, which read and then write. Furthermore, these transfer times can be used even if the architecture implements an explicit block move instruction that does not directly involve process registers; in this case, the transfer time may be parameterized by the length of the move since a block move may be more efficient for longer transfers. The transfer times, per byte, are MT_{sp}, shared memory to private, MT_{ss}, shared to shared, MT_{ps}, private to shared, and MT_{pp}, private to private.

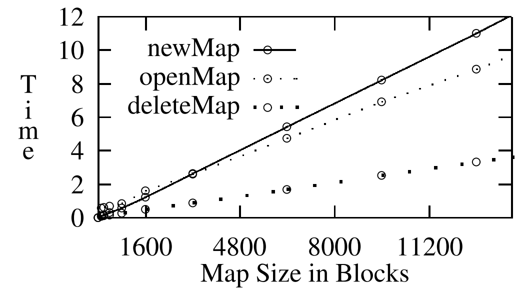
3.1 Disk Transfer Time

Modelling disk transfer is complex since it is a function of the access pattern due to the inherent sequentiality of the components of a disk access. The nature of join algorithms is such that data is clustered into contiguous bands on the disk during certain parts of an algorithm. Intense I/O occurs in a band followed by similar I/O occurring in the next band and so on. This clustering is modelled by measuring the average cost (per block) of sequentially accessing bands in which random access occurs, over a large area of disk. The size of the disk area is irrelevant; it only has to be large enough to obtain an average access time for the band size. The layout of data on disk is always given to explain the band size in our algorithms.

In general, the disk transfer time function, *dtt*, has two arguments: the unit of data transfer, and the span, in blocks, over which random disk accesses take place, i.e. the size of the band. In our work, the first argument is always B, the virtual memory page size; therefore, the first argument is dropped from all of our subsequent formulas. Fig. 1(a) shows the average time to transfer a block (4K in our experiments) to or from disk with respect to a given band size. When the band size is one block, access is sequential; when the band size is greater, access is random over that area. Thus, average time increases as the band size increases. One curve is for random reading in a band (no duplicates); the other curve is for a random writing in the band (no duplicates). One might expect the read and write times to be identical. However, while a read page fault must cause an immediate I/O operation, writing dirty pages can be deferred allowing for the possibility of parallel I/O and optimization using shortest seek-time scheduling algorithms. Thus, writes, on average, cost less than reads. The two curves are used to interpolate disk transfer times for reading, dtt_r, and writing, dtt_w, respectively. Both dtt_r and dtt_w are machine dependent and must be measured for the specific machine/disk combination doing the join.



(a) Disk Transfer Time (in msecs per block)



(b) Memory Mapping Setup Time (in secs)

Figure 1: Measured Machine Dependent Functions (Sequent Symmetry/Dynix using Fujitsu M2344K and M2372K disk drives)

3.2 Memory Mapping Setup

The cost of three fundamental memory mapping operations, namely, creating a mapping for a new area of disk, establishing a mapping to an existing area of disk, and destroying a mapping as well as its data in an existing area of disk, is modelled by three measured functions, *newMap*, *openMap* and *deleteMap*. Each of these functions takes the size of the mapping as an argument.

Fig. 1(b) shows the measured values of these three functions for our experiments. All mapping costs increase with size since constructing the page table and acquiring disk space increases linearly with the size of the file mapped. New mappings are more expensive than existing mappings since new disk space must be acquired. Deleting is the least expensive as only the storage for the page table and disk space need to be freed.

4 Parallel Pointer-Based Join Algorithms

We design and analyze the parallel pointer-based versions of nested loops, sort-merge, and a variation of Grace join algorithms. We consider the joining of a relation R with S. In pointer-based join algorithms, the join attribute is a virtual pointer to objects in S, which is ideal for our memory mapped environment and results in significant performance advantages. A virtual pointer provides an implicit ordering of objects in S, which is exploited to eliminate the usual sorting or hashing of S in sort-merge and hash-based joins, respectively.

We assume that S is initially partitioned on D disks into equal-sized partitions $S_1, \ldots, S_D$ and that the containing partition for an object of S can be computed, in time *map*, from a pointer to that object. We further assume that the join attributes are randomly distributed in R, which is also divided into equal-sized partitions $R_1, \ldots, R_D$. Finally, each relation is managed by a process (*Rproc* and *Sproc*) which is aware of the structure of the relations, and *Rproc* is capable of carrying out the join itself.

The following parameters are defined for various relations and their subsets. $|X|$ denotes the number of objects in X and P_X is the number of pages in X. r, s denote the size of a single object in R and S respectively.

For our algorithms, private memory is viewed as being divided into D pieces, where the ith piece is associated with partition R_i. We describe the algorithm as it progresses on the ith piece, with the understanding that work on the remaining $D-1$ pieces is progressing in an analogous fashion in parallel. Our experiments use D processors each for R and S to achieve maximum parallelism.The partitions of R are further divided into sub-partitions based on the partitions of S to which the join attributes point. The subset of R_i with join attributes residing in partition S_j is called $R_{i,j}$. R_{S_j} denotes the set of all objects in R that have pointers to objects in S_j, i.e., $R_{S_j} = \bigcup_{i=1}^{D} R_{i,j}$. This substructure is illustrated in the figures for subsequent algorithms. For a given i, the $R_{i,j}$ sub-partition may have some skew in size since sub-partitions may contain more references to some S_j and fewer to others; the amount of skew is defined as $skew = \max_j\{|R_{i,j}|/(|R_i|/D)\}$. Skew is important as it affects the performance of certain algorithms.

A few more necessary parameters are defined with each specific algorithm. Finally, since every algorithm forms and outputs the same join, we do not count the time to do this in the analysis, nor do we assume that the join results are generated in any particular order.

In our analyses of join algorithms we compute quantities of time that can be summed to give the total elapsed time for $Rproc_i$. Since there is little or no contention during the D-fold parallelism, the total elapsed time for $Rproc_i$ also represents the total time for the entire join.

While it is convenient to speak of data being read or written in the algorithms, input and output is not explicitly requested by our algorithms. When we speak of reading a block of data, the implementation actually accesses a location in virtual memory mapped to that block. If the block is not in primary memory, it is read in by means of a page fault; otherwise, no disk access takes place. Similarly, when we speak of writing a file, no explicit action occurs in the implementation; the writing of a (dirty) block of data takes place when that page is replaced by the operating system. These actions are similar to what occurs in an explicitly managed buffer pool, where objects are fetched from already read buffers and written only when the buffer is written, albeit with more user control.

5 Parallel Pointer-Based Nested Loops

Nested loops join works by sequentially traversing R, and for each R-object, accessing the S-object pointed to by the join attribute; R is called the outer relation and S the inner. The resulting random accesses to S make nested loops inefficient. A naive parallel version may partition R and S so that the R_i partitions can perform the join in parallel, accessing different S_j partitions simultaneously. However, parallelism in this case is inhibited by contention when several R_i reference the same S_j; this contention can be reduced or eliminated.

In the traditional nested loops join, the smaller of the two relations is used as the inner relation so that it can be kept in the buffer pool. In our system, S is always the inner relation unless S objects contain back pointers to R.

5.1 Algorithm

For each partition R_i in parallel, the algorithm operates in two passes. In pass 0 (see fig. 2), R_i is read, one object at a time, into the private memory of $Rproc_i$. In terms of actual I/O, this translates to reading R_i in chunks of the virtual memory page size, B. In fig. 2, an object is represented by a tuple $(MAP(sptr), sptr)$, where $MAP(sptr)$ is the S partition containing the object pointed to by $sptr$. For each object in R_i, the S partition is computed from the join attribute and the object is copied (written) to a sub-partition inside of a temporary area RP_i, which is mapped onto the same disk as partition R_i. Hence, all the R-objects in R_i that point to an object in S_j are grouped together in sub-partition $RP_{i,j}$. This sub-partitioning (mostly) eliminates disk contention in the next pass.

As an optimization, the objects in R_i that point to objects in S_i are immediately joined by extracting the join pointer, and having $Sproc_i$ read the corresponding S object. $Sproc_i$ dereferences the join attribute resulting in a read of the page of S_i containing the object, if that page is not already in memory, and makes the S object available for the join by putting it into shared memory. $Rproc_i$ then does the join. As a further optimization, the requests for objects from S_i are grouped into a buffer of size G to reduce context switches between $Rproc_i$ and $Sproc_i$.

Instead of putting RP_i in its own segment, managed by another process, it is made part of the storage for $Rproc_i$. That is, R_i is located at the lowest address of the $Rproc_i$ segment and storage for RP_i is located after the storage for R_i. Hence, both R_i and RP_i are mapped to the private memory of $Rproc_i$. This organization eliminates the costs of segment-to-segment transfer, namely copying data through shared memory. It also eliminates the cost of creating and managing an additional process for RP_i. The drawback of this optimization is that the maximum size of R_i is approximately half of the maximum address space size.

Pass 1 (see fig. 2) eliminates disk contention by staggering access to S_i through a series of $D-1$ phases without synchronizing the phases. In phase t ($t = 1, 2, \ldots D-1$), $RP_{i,offset(i,t)}$ is joined with $S_{offset(i,t)}$, where $offset(i,t) = ((i+t-1) \bmod D) + 1$. $Rproc_i$ loops over objects in $RP_{i,offset(i,t)}$ in private memory; for each one, it extracts the

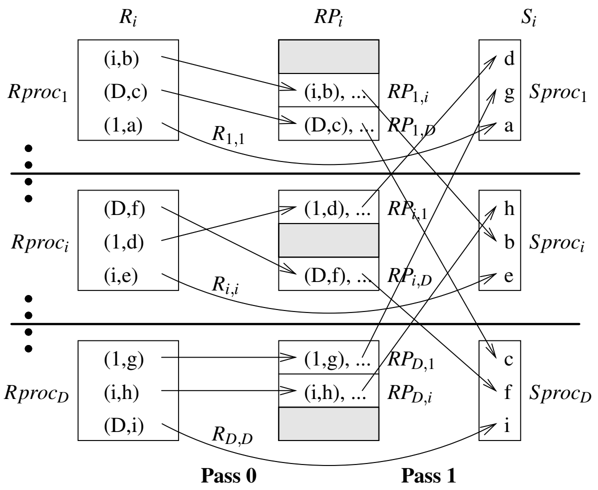


Figure 2: Parallel Pointer-Based Nested Loops

join pointer and asks $Sproc_{offset(i,t)}$ for the corresponding S object. Due to the offset, S_j is only accessed by one $Rproc_i$ in any one phase, assuming no skew. In the presence of skew, there are different numbers of objects in each $RP_{i,j}$, so there may be some contention when multiple $Rproc_i$ access the same S_j.

Since we assume a random distribution of join attributes, *skew* is very close to 1.0. As a result, no synchronization is used after each phase of pass 1 for all the $Rproc_i$; any contention is insignificant, as was verified by running experiments with synchronization after each phase of pass 1. In the best case, there was a 0.5% decrease in I/O and total time due to reduced contention.

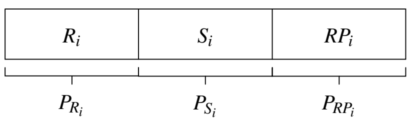
5.2 Parameter Choices

M_{Rproc_i} should be large enough to hold, in pass 0, at least one block of the input R_i and at least one block for each $RP_{i,j}$. Since S_i is being read randomly, M_{Sproc_i} should be as large as possible. G should be large enough to avoid many context switches between $Rproc_i$ and $Sproc_i$, but small enough so that the volume of pending requests does not force important information out of memory. The implementation used a value of B for G.

5.3 Analysis

Given $|R_i| = |R|/D$ and $|R_{i,i}| = |R_i|/D \cdot skew = |R|/D^2 \cdot skew$, for the largest of $R_{i,i}$, then $|RP_i| = |R_i| - |R_{i,i}| = (|R|/D)(1 - skew/D)$. R_i is *not* adjusted by *skew* since there is no synchronization between phases in this algorithm; in essence, the skew in $RP_{i,j}$ is compensated for by the additional parallelism resulting from the lack of synchronization among the $Rproc_i$ between passes 0 and 1.

In pass 0, R_i is read sequentially, RP_i is written (mostly) randomly, and S_i is read randomly. The disk layout of the three partitions is:



Since each partition is accessed, the band size of disk arm movement, $BandSize_{pass0}$, in the worst case, is the total size of all partitions:

$$P_{R_i} + P_{S_i} + P_{RP_i} = \frac{P_R}{D} + \frac{P_S}{D} + \left(\frac{P_R}{D} - \frac{P_R}{D^2} \cdot skew\right).$$

As well, since random reads and writes are interspersed on the same disk, all *dtt* costs are for random I/O (i.e., it does not matter that some objects are read sequentially). The disk transfer times for R_i and RP_i are $P_{R_i} \cdot dtt_r(BandSize_{pass0})$ and $P_{RP_i} \cdot dtt_w(BandSize_{pass0})$.

$|R_{i,i}|$ S-objects are read randomly from S_i, one object at a time, during the join, but some of those objects may be in memory already when requested. We use a result of Mackert and Lohman [23] to approximate the number of page faults, which corresponds to disk transfers. Their paper derives the following approximation: given a relation of N tuples over t pages, with i distinct key values and a b-page LRU buffer, if x key values are used to retrieve all matching tuples, then the number of page faults is

$$Y_{lru}(N,t,i,b,x) = \begin{cases} t(1-q^x) & \text{if } x \le n \\ t[(1-q^n) + p(x-n)q^n] & \text{if } x > n \end{cases}$$

where $n = \max\{j : j \le i, t(1-q^j) \le b\}$ and $q = 1 - p = (1 - 1/\max(t,i))^{N/\min(t,i)}$.

Assuming the references to S are randomly distributed in R, the disk transfer time for S_i in pass 0 is

$$Y_{lru}\left(|R_{S_i}|, P_{S_i}, |R_{S_i}|, \frac{M_{Sproc_i}}{B}, |R_{i,i}|\right) \cdot dtt_r(BandSize_{pass0}).$$

In pass 1, RP_i is read sequentially and S_i is read randomly. Since only S_i and RP_i are used, the band size of disk arm movement, in the worst case, is the total size of both partitions: $BandSize_{pass1} = P_{S_i} + P_{RP_i}$. As well, since random reads and writes are interspersed on the same disk, all *dtt* costs are for random I/O. The disk transfer times for RP_i and S_i are $P_{RP_i} \cdot dtt_r(BandSize_{pass1})$ and

$$Y_{lru}\left(|R_{S_i}|, P_{S_i}, |R_{S_i}|, \frac{M_{Sproc_i}}{B}, |RP_i|\right) \cdot dtt_r(BandSize_{pass1}).$$

Further, in pass 0, each object of R_i is moved once, either to RP_i or to shared memory for the join, and appropriate objects of S_i are moved to shared memory for the join. The transfers from R_i to RP_i or shared memory are simple memory transfers among areas of $Rproc_i$'s memory due to the organization of $Rproc_i$'s memory (see section 5.1). The corresponding data transfer cost is:
$|RP_i| \cdot r \cdot MT_{pp} + |R_{i,i}| \cdot (r + sptr + s) \cdot MT_{ps}$.

The transfers from S_i require a data movement from $Sproc_i$'s private memory to shared memory so that an object can be accessed by $Rproc_i$; this requires two context switches, from $Rproc_i$ to $Sproc_i$ and back again so that $Sproc_i$ can perform the transfer. To optimize context switching, shared memory of size G is used (see section 5.1). During the sequential pass of R_i, objects for $R_{i,i}$ and their join attributes (i.e., the S-pointers) are placed into this buffer until there is only room for the corresponding S_i objects. While the S-pointer is available in the R object, it is copied so that $Sproc_i$ does not have to know about the internal structure of R objects. The buffer is then given to $Sproc_i$ to copy the corresponding objects into the remaining portion of the buffer. The objects in the buffer can now be joined. The buffer reduces the number of context switches to $Sproc_i$. The alternative is to join each individual R object when found during the sequential scan, which results in a context switch to $Sproc_i$ for each object.

In pass 1, each object of RP_i is moved once to shared memory, and appropriate objects of S_i are moved to shared memory for the join in a total time of $|RP_i| \cdot (r + sptr + s) \cdot MT_{ps}$. The buffering technique employed in pass 0 is also used in pass 1 to retrieve S-objects. The context switching costs for pass 0 and 1 are $g(|R_{i,i}|)$ and $g(|RP_i|)$ respectively, where $g(h) = 2 \cdot CS \cdot \lceil h / \lfloor G/(r + sptr + s) \rfloor \rceil$. The cost of mapping the join attributes to their S partitions in pass 0 is $|R_i| \cdot map$.

Finally, the setup cost (see section 3.2) for mapping R_i, S_i and RP_i is $D \cdot (openMap(P_{R_i}) \ +openMap(P_{S_i}) \ +newMap(P_{RP_i}))$. The setup time is multiplied by D since manipulating a mapping is a serial operation.

6 Parallel Pointer-Based Sort-Merge

In nested loops, the random access of S slows down the join. Sort-merge changes the random access to a single sequential scan of S, resulting in a significant performance gain. While Shapiro's sort-merge [31] assumes only two passes, we permit multiple passes, writing out full records at each pass. Also, as noted earlier, the use of S-pointers as the join attribute makes sorting of S_i unnecessary.

6.1 Algorithm

The first two passes of the parallel sort-merge algorithm are the same as in parallel nested loops (see sec. 5.1) except for one difference; in nested loops, $R_{i,i}$ and $RP_{i,j}$ are joined with S_i, whereas in sort-merge, $R_{i,i}$ and $RP_{i,j}$ are written out to R_{S_i}. Fig. 3 shows these two passes for sort-merge.

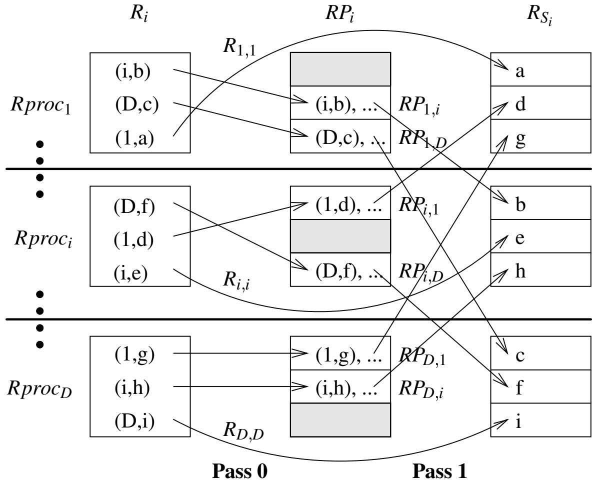


Figure 3: Parallel Pointer-Based Sort-Merge

Once the R_{S_i}'s have been formed, the sequential sort-merge algorithm is executed on each partition. The algorithm proceeds by first sorting, in parallel, all R_{S_i} with respect to the join attributes to allow sequential access to S_i. Sorting of R_{S_i} is done using multi-way merge sort, with the aid of a heap and with intermediate runs stored on disk. In the final pass, S_i is read in sequentially to perform the join.

As in nested loops, data movement is optimized by combining several partitions in $Rproc_i$'s segment. That is, R_i is located at the lowest address of the $Rproc_i$ segment, storage for RP_i is located after that, and then *all* partitions for the R_{S_j}. Hence, all these partitions are in the private memory of $Rproc_i$. The saving in data transfers through shared memory is significant and is possible since RP_i and the R_{S_j} are temporary areas where the data is manipulated as objects without the need to dereference internal pointers. The drawback is that the maximum size of R_i is approximately $D+1$ times less than the maximum address space size. If this optimization poses a problem, the R_{S_j} can be separate segments; data can be copied to them through shared memory using a buffer.

Our design and analysis introduce a number of parameters, some chosen by the programmer, and some specified by the implementation. The programmer must choose $IRUN$, the length of a run created from unsorted data from pass 1, and $NRUN$, the number of runs to be merged in a merging pass. In pass 2, $IRUN$ R-objects are read in and a heap of pointers to these objects is created. Heapsort is applied to this heap of pointers and then the sorted list of pointers is used to sort, in place, the corresponding R-objects. The resulting sorted run of $IRUN$ R-objects is (eventually) written out to disk. These actions are repeated to sort successive runs until all of R_{S_i} has been processed.

On subsequent merging passes, groups of $NRUN$ sorted runs are merged using delete-insert operations on a heap of $NRUN$ pointers. The heap always contains pointers to the next unprocessed element from each sorted run; when a pointer is deleted from the heap, the corresponding object is moved to the output run, and a pointer to the next object from the input run that contained the moved object is inserted into the heap.

On the last merging pass, instead of writing out the merged R-objects, the corresponding objects from S_i are read sequentially and the join computed. The reading of the objects from S_i is accomplished, as in nested loops, by means of a shared memory buffer of size G.

6.2 Parameter Choices

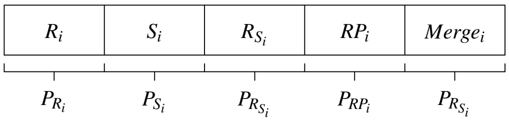
$IRUN$ is chosen to be the maximum so that an entire run, plus space for the heap of pointers, fits in memory, i.e., $IRUN = \lfloor M_{Rproc_i}/(r + hp) \rfloor$ where hp is the size, in bytes, of an element in the heap of pointers.

Ideally, merging of runs requires at least one page of memory for each run; otherwise excessive thrashing occurs since pages are replaced before they are completely processed. In reality, with this minimum memory, pages are replaced prematurely since the LRU paging scheme makes the wrong decisions when replacing a page during the merging passes. That is, when objects in an input page have been processed, the page is no longer needed, but it must age before it is finally removed; during the aging process, a page that is still being used for the output runs gets paged out, resulting in additional I/O. In our implementation, we avoid the problem by reducing the value of $NRUN$, which is chosen to be $M_{Rproc_i}/(3 \cdot B)$ during all but the last pass (denoted $NRUN_{ABL}$), and $M_{Rproc_i}/(2 \cdot B)$ during the last pass. In other words, memory is underutilized to compensate for this anomaly so that the program behaves more consistently. The amount of underutilization is based on an approximation of the working set of the program during these passes.

6.3 Analysis

Given $|R_i| = |R|/D$ and $|R_{i,i}| = |R_i|/D \cdot skew$, for the largest of $R_{i,i}$, then $|RP_i| = |R_i| \cdot skew - |R_{i,i}| = (|R| \cdot skew/D)(1-1/D)$. R_i *is* adjusted by *skew* since there is synchronization between phases in this algorithm, therefore the worst case must be considered for each individual pass.

In pass 0, R_i is read sequentially, RP_i is written (mostly) randomly, and R_{S_i} is written sequentially. Disk layout is:



resulting in the band size of disk arm movement, $BandSize_{pass0}$, in the worst case, of

$$P_{R_i} + P_{S_i} + P_{R_{S_i}} + P_{RP_i} = \frac{P_R}{D} + \frac{P_S}{D} + \frac{P_R}{D} + \left(\frac{P_R}{D} - \frac{P_R}{D^2}\right) \cdot skew$$

The disk transfer times for R_i, R_{S_i} and RP_i are $P_{R_i} \cdot dtt_r(BandSize_{pass0})$, $P_{R_{S_i}} \cdot dtt_w(BandSize_{pass0})$ and $P_{RP_i} \cdot dtt_w(BandSize_{pass0})$, respectively.

In pass 1, RP_i is read sequentially, and R_{S_i} is written sequentially, giving $BandSize_{pass1} = P_{R_{S_i}} + P_{RP_i}$. The disk transfer times for R_{S_i} and RP_i are $P_{R_{S_i}} \cdot dtt_w(BandSize_{pass1})$ and $P_{RP_i} \cdot dtt_r(BandSize_{pass1})$ respectively. All *dtt* formulas are for random I/O due to wide fluctuations in the disk arm between regions read or written sequentially.

In pass 0, each object of R_i is moved once within $Rproc_i$'s segment, either to RP_i or to R_{S_i}, at a cost of $|R_i| \cdot r \cdot MT_{pp}$. In pass 1, each object of RP_i is moved once within $Rproc_i$'s segment to the appropriate R_{S_i} at a cost of $|RP_i| \cdot r \cdot MT_{pp}$. The mapping cost for pass 0, which generate an S partition from an S-pointer, is $|R_i| \cdot map$.

In pass 2 (the heap-sorting pass), runs of size *IRUN* objects from R_{S_i} are sequentially read in and sorted in place. Since there is no explicit writing, the previous sorted run is written back by the operating system as the pages age with mostly random writes. This results in a disk band size that is twice the size of a sort run: $2 \cdot (r \cdot IRUN/B)$. The disk transfer times for reading R_{S_i} and writing back the sorted runs are $P_{R_{S_i}} \cdot dtt_r(2r \cdot IRUN/B)$ and $P_{R_{S_i}} \cdot dtt_w(2r \cdot IRUN/B)$, respectively.

We define *compare* to be the amount of time $Rproc_i$ requires to compare two elements in a heap of pointers to R-objects, stored in memory. Similarly, *swap* is the amount of time to swap two heap elements stored in memory, and *transfer* is the amount of time to move an element to or from the heap. This does not count operations necessary to restore heap discipline; those are computed separately.

In order to heap-sort each individual run, an array of pointers to the *IRUN* R-objects in memory is converted into a heap using Floyd's heap construction algorithm (see [16, 15]). The heapsort method outlined in [29] is then used with a modification suggested by Munro that allows the heapsort to complete, in the average case, with approximately $N \log N$ comparisons and transfers. The creation of the heap takes time $1.77 \cdot |R_{S_i}| \cdot (compare + swap/2) + |R_{S_i}| \cdot transfer$ while the cost of heap-sorting the heap by repeated deletion of minima is $|R_{S_i}| \cdot \log IRUN \cdot (compare + transfer)$. A further cost of $|R_{S_i}| \cdot r \cdot MT_{pp}$ is required to move the actual R-objects in place based on the sorted list of pointers.

The choice of *IRUN* and $NRUN_{ABL}$ in turn determines *NPASS*, the number of merging passes, and *LRUN*, the number of runs on the last pass. The value of *NPASS* is

$$\max\left\{ j : j \geq 1, \left\lceil \frac{|R_i|}{IRUN \cdot (NRUN_{ABL})^{j-1}} \right\rceil \leq NRUN \right\}$$

and that of LRUN is $\lceil |R_i|/(IRUN \cdot (NRUN_{ABL})^{NPASS-1}) \rceil$.

In the third and subsequent passes, groups of $NRUN_{ABL}$ (or *LRUN* in the last pass) input runs are read in, merged into one, and written out. The storage for R_{S_i} and $Merge_i$ alternate as source and destination of these runs. The disk band size during all but the last pass is $BandSize_{ABL} = P_{R_{S_i}} + P_{RP_i} + P_{Merge_i}$, and during the last merging/joining pass is $BandSize_{Last} = P_{S_i} + P_{R_{S_i}} + (P_{RP_i} + P_{Merge_i}) \cdot (NPASS+1) \bmod 2$.

The disk transfer time, except for the last pass, for reading and writing R_{S_i} and $Merge_i$ $NPASS-1$ times are $P_{R_{S_i}} \cdot dtt_r(BandSize_{ABL}) \cdot (NPASS-1)$ and $P_{R_{S_i}} \cdot dtt_w(BandSize_{ABL}) \cdot (NPASS-1)$, respectively. During the last pass, respective I/O costs for R_{S_i} and S_i are $P_{R_{S_i}} \cdot dtt_r(BandSize_{Last})$ and $P_{S_i} \cdot dtt_r(BandSize_{Last})$.

During the merge, except for the last pass, the delete-insert operation [15, p. 214] is used on a heap of size $NRUN_{ABL}$ and the heap operations take time $(g(NRUN_{ABL}) + 2 \cdot transfer) \cdot |R_{S_i}| \cdot (NPASS-1)$, where $g(h) = (2 \cdot compare + swap) \cdot ((h+1) \cdot k - \lfloor h/2 \rfloor - 2^k)/h$ and $k = \lfloor \log h \rfloor + 1$. The size of the heap during the last merge pass is *LRUN* and the heap operations take time $(g(LRUN) + 2 \cdot transfer) \cdot |R_{S_i}|$. The respective data transfer cost during the $NPASS-1$ merge passes and the last merge pass are $|R_{S_i}| \cdot r \cdot MT_{pp} \cdot (NPASS-1)$ and $|R_{S_i}| \cdot (r + sptr + s) \cdot MT_{ps}$, with the corresponding context switching time of $2 \cdot CS \cdot \lceil |R_{S_i}|/\lfloor G/(r+sptr+s) \rfloor \rceil$. Finally, the setup cost for mapping R_i, S_i, R_{S_i}, RP_i and $Merge_i$ is $D \cdot (openMap(P_{R_i}) + openMap(P_{S_i}) + newMap(P_{R_{S_i}}) + newMap(P_{RP_i}) + newMap(P_{S_i}))$. The setup time is multiplied by D since manipulating a mapping is a serial operation. An additional cost of $(deleteMap(P_{S_i}) + newMap(P_{S_i})) \cdot (NPASS-1)$ is incurred in all but the last merge passes to swap the source and destination areas.

7 Parallel Pointer-Based Grace

Sort-merge improves the performance of the join by sorting R_i by the S-pointer, which allows sequential reading of S_i. However, sorting is an expensive operation.

Hash-based join algorithms replace the sort with hashing to improve performance further. As a representative example of the hash-based join algorithms, we have chosen to model a parallelized pointer-based version of the Grace algorithm. Modelling of other more modern hash-based join algorithms will be done in future work.

As with sort-merge, the ordering property of the S-pointers makes it unnecessary to hash S_i. By carefully designing the hash algorithm, it can be ensured that each hash bucket contains monotonically increasing locations in S_i, so that S_i can be read sequentially.

7.1 Algorithm

The first two passes of the Grace algorithm, shown in fig. 4, are the same as in parallel nested loops, except for one difference; in nested loops, R-objects are joined with S_i, whereas in Grace, the join attributes (i.e., the S-pointers) from R-objects are hashed into one of K sub-partitions (or buckets) that make up R_{S_i}. The value of K is chosen by the programmer based on the amount of memory available. We refer to the jth sub-partition of R_{S_i} as $B_{S_i,j}$, i.e., $R_{S_i} = \bigcup_{j=1}^{K} B_{S_i,j}$.

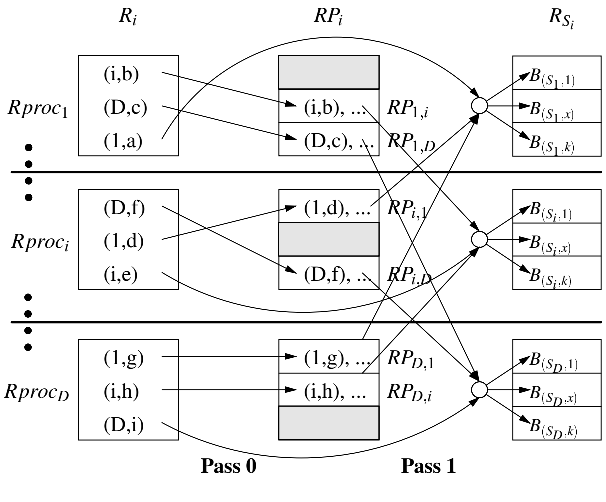


Figure 4: Parallel Pointer-Based Grace

In pass 1, $Rproc_i$ reads $R_{i,j}$ (for all $j \neq i$), one R-object at a time, and hashes each object into one of the K sub-partitions of R_{S_j}. As before, the reading and hashing of $R_{i,j}$'s in pass 1 takes place in phases to eliminate contention for the disks. At the end of pass 1, each R_{S_i} contains K sub-partitions with hashed R-objects. The hash function is chosen so as to cluster R-objects by the value of their join attributes. Therefore, $B_{S_i,j}, j = 1, \ldots K-1$, contains R-objects with join attributes smaller than that of any R-object in $B_{S_i,j+1}$.

In pass $1+j$ ($j = 1, 2, \ldots K$), for every i in parallel, $B_{S_i,j}$ is read in, and the value of the join attribute in each object is used as input to another hash function that further refines the partitioning given by the first hash function. The range of this hash function is $TSIZE$, a parameter chosen by the programmer. Once all of $B_{S_i,j}$ has been hashed into this in-memory hash table, the table is processed in order. Common references to objects in S_i (i.e., references that result in a collision when hashed) are in the same hash chain. If we assume that there are no more than M_{Sproc_i}/s different references to objects in S_i in any one hash chain during the processing of that hash chain, all objects from S_i needed during this processing can fit in memory; hence each object referenced from S need only be read once in order to perform the join. The reading of the objects from S_i is accomplished using a shared memory buffer of size G.

7.2 Parameter Choices

During pass $1+j$, $j = 1, 2, \ldots K$, $Rproc_i$ reads each R-object in $B_{S_i,j}$ into a memory resident hash table. The value of K should be chosen such that each $B_{S_i,j}$ along with its associated hash table overhead fits entirely in memory.

$TSIZE$ should be small enough to avoid excessive hash-table overhead due to underutilization of memory and large enough to ensure short individual hash chains. Theoretically, the minimum amount of memory that needs to be made available to each $Rproc_i$, in pass 0, to avoid thrashing is $D + \lceil \sqrt{fuzz \cdot |R_i| \cdot r/B} \rceil$ blocks, where $fuzz$ makes room for the hash table overhead. In reality, even this threshold memory results in thrashing, because the working set for the algorithm is greater than the theoretical threshold memory and the LRU paging scheme then makes the wrong decision, removing useful pages prematurely. See [31, 34] for more discussion on this problem. In the next section, we derive an approximation for the amount of extra I/O that takes place when memory is insufficient.

7.3 Analysis

The disk band sizes during pass 0 and pass 1 are $BandSize_{pass0} = P_{R_i} + P_{S_i} + P_{R_{S_i}} + P_{RP_i}$ and $BandSize_{pass1} = P_{R_{S_i}} + P_{RP_i}$, where P_{RP_i} is the same size as in sort-merge because there is synchronization between phases. Pass 0 involves reading objects from R_i, one object at a time, and writing each object to either RP_i or to one of the K buckets in R_{S_i}. The corresponding costs are $P_{R_i} \cdot dtt_r(BandSize_0)$, $P_{RP_i} \cdot dtt_w(BandSize_0)$ and $(P_{R_{i,i}} + K) \cdot dtt_w(BandSize_0)$. The number of pages written to R_{S_i} has been increased by K to account for the fact that objects read from $R_{i,i}$ are hashed into K buckets in R_{S_i}. The additional costs incurred in pass 0 include $|R_i| \cdot map$ to map the join attributes to their corresponding S partition, $|R_{i,i}| \cdot hash$ to hash the $R_{i,i}$ objects into one of K R_{S_i} buckets and $|R_i| \cdot MT_{PP}$ to move the R_i objects in private memory to either RP_i or R_{S_i}.

We use an urn model to derive an approximation for the amount of extra I/O that takes place due to lack of memory in pass 0. In pass 0, R-objects from R_i are placed in one of $RP_{i,j}$ or in one of the K buckets of R_{S_i}. Once hit, a bucket page is replaced when there are M_{Rproc_i}/B references to newer pages before it is hit again; the probability of hashing t further objects without a second hit is $(1-1/K)^t$. At any given time, some of the pages in memory are partially filled or read pages (*current pages*) and some pages have been completely processed or filled (*fill events*) but which stay around since they are recently accessed. We assume that the D current pages for R_i and $RP_{i,j}$ stay in memory until processed completely; these pages are processed at a much faster rate than the pages in R_{S_i}.

For convenience, we divide the hashing of objects after a hit into epochs; the first α_0 objects, the next α_1, and so on.

The number of fill events that have occurred at the beginning of epoch q is a random variable which we approximate by a constant depending on q. Since the page replacement algorithm prefers clean pages over dirty pages, we ignore the fill events caused by the processing of R_i. The fill rate for $RP_{i,j}$ is $\lceil (D-1)/|B| \rceil$, and for R_{S_i} is $\lceil 1/(K \cdot |B|) \rceil$, the latter being negligible. Therefore, the number of fill events is $\lceil H_j \cdot (D-1)/|B| \rceil$, where $H_j = \sum_{n=0}^{j-1} \alpha_n$ is the number of objects hashed at the beginning of epoch j.

The probability that at most $\lceil H_j \cdot (D-1)/|B| \rceil + D$ buckets are not hit by the beginning of the epoch (denoted p_j) multiplied by the probability that a page gets hit again during epoch j, denoted y_j, is the probability that the page is not present in memory during a second hit in epoch j. Summing over all epochs and multiplying by $|R_{i,i}|$ gives an approximation to the expected number of times a page of R_{S_i} gets replaced prematurely.

The probability p_j can be computed by reference to Johnson and Kotz [19, p.110], who state that the probability, $Pr[X=k]$, of exactly k urns being empty after n balls are randomly placed into m urns is

$$\binom{m}{k} \cdot \left(1 - \frac{k}{m}\right)^n \cdot \sum_{j=0}^{m-k-1} \binom{m-k}{j} (-1)^j \left(1 - \frac{j}{m-k}\right)^n .$$

Every premature replacement necessitates one extra write (to replace the page) and one extra read (when the page is referenced again) for a total cost of reading and writing of $|R_{i,i}| \cdot \sum_{j \geq 1} (p_j \cdot y_j)$ blocks. For our computations we used size K for the first epoch and 1 for the rest.

In pass 1, objects in $RP_{i,j}$ are read, one object at a time, and each object is hashed into one of the K buckets in R_{S_j}. The costs of reading RP_i and writing R_{S_i} are $P_{RP_i} \cdot dtt_r(BandSize_1)$ and $(P_{RP_i} + K) \cdot dtt_w(BandSize_1)$, respectively. Once again, the number of pages written to R_{S_j}'s has been increased by K. It takes a further time of $|RP_i| \cdot MT_{PP}$ to move the objects in private memory.

After pass 1, the subsequent reading of the partitioned R_{S_i}, one bucket at a time, and the corresponding S_i objects requires time $(P_{R_{S_i}} + P_{S_i}) \cdot dtt_r(P_{R_{S_i}}/(K/2)$. The band size for dtt_r is chosen to be half the size, in blocks, of the objects that fit in the hash table. This is done to approximate the actual behaviour, which is to read sequentially objects from a sub-partition of R_{S_i} followed by the corresponding objects in S_i and so on.

Each object in R_{S_i} is hashed once during the processing of each bucket, for time $|R_{S_i}| \cdot hash$. The cost of transferring objects to shared memory is $|R_{S_i}| \cdot MT_{PS} \cdot (r + sptr + s)$ with the corresponding context switching time of $2 \cdot CS \cdot \lceil |R_{S_i}| / \lfloor G/(r + sptr + s) \rfloor \rceil$.

Finally, the setup costs for mapping R_i and S_i for reading, creating the new mappings for R_{S_i} and RP_i in pass 0 and setting up R_{S_i} for reading in pass 1 is $D \cdot (openMap(P_{R_i}) + openMap(P_{S_i}) + newMap(P_{R_{S_i}} + P_{RP_i}) + openMap(P_{R_{S_i}}))$.

8 Model Validation

To validate the model and the analysis presented earlier, experiments were run that performed full joins on two relations with 102,400 objects each. The objects in each relation were 128 bytes. R and S were partitioned across 4 disks with one R and one S partition on each disk.

All experiments were run on a Sequent Symmetry with 10 i386 processors, which uses a simple page replacement algorithm (see [11] for details of our test bed). The file system was adjusted so all virtual memory I/O was done in 4K blocks. The execution environment was strictly controlled so the results were not influenced by any outside activity. The amount of memory for the experiment's address space and the global cache were tightly controlled.

Fig. 5 shows the predicted and measured elapse times for running the various join algorithms with varying amounts of memory available. The discontinuities in the sort-merge graph occur when additional merging phases are required. The curve in the Grace graph at low memory levels results from thrashing caused by the page replacement algorithm.

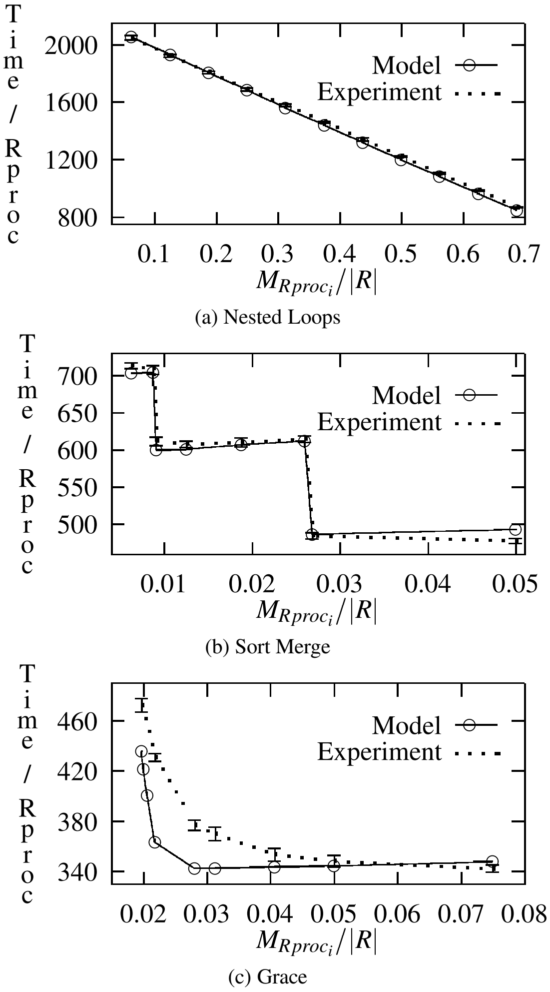


Figure 5: Experimental Results

As is evident from the graphs, our model does an excellent job of predicting performance for the various join algorithms in almost all conditions. In particular, there is a close match between prediction and actual performance for nested loops and sort-merge. All the experiments were repeated several times to make sure that the results were consistent, accurate and reproducible. For Grace, our approximation for I/O caused by thrashing at low memory levels is reasonably accurate; there is scope for further refinement of this approximation. A major part of the difference between prediction and actual behavior at low memory levels comes from the overhead introduced by the particular replacement strategy used by the Dynix operating system. Modelling this aspect of the page replacement scheme will be done in future work.

9 Conclusions and Further Work

We have designed and validated a quantitative analytical model for database computing in a particular kind of memory mapped environment. We have successfully used our model to make accurate predictions about the real time behaviour of three different parallel join algorithms, namely, nested-loops, sort-merge and a variation of Grace. Future studies with the model will include speedup and scaleup experiments, changing the nature of the joining relations and a comparative analysis of various algorithms. Our methodology allows the use of virtual pointers as the join attributes, which in turn introduces significant performance gain by eliminating the need to sort/hash one of the two relations. Our analysis of the join algorithms also highlighted an inherent drawback in single level stores: the lack of control over buffer management on the part of the database application results in incorrect decisions being made at times by the underlying page replacement strategy. While accepting this inefficiency, we have demonstrated two approaches to achieving predictable behaviour, an essential property in a database system. With single-level stores becoming more common, it is our hope that future research and development in operating system architecture will make it feasible for database applications to exercise more control over the replacement strategies used (see [7]). There is scope for further improvement in the design of our model, especially in the modelling of the underlying paging behaviour. Future work will involve extending our model to other memory mapped environments allowing us to perform comparative studies. It will also be an interesting exercise to explore the applicability of our model to traditional join algorithms.

References

[1] Aggarwal, A. et al. A Model for Hierarchical Memory. In *ACM STOC*, pp. 305–314, May 1987.

[2] Aggarwal, A. and Chandra, A. K. Communication Complexity of PRAMs. In *ICALP*, pp. 1–17, 1988.

[3] Aggarwal, A. et al. Hierarchical Memory with Block Transfer. In *IEEE FOCS*, pp. 204–217, 1987.

[4] Aggarwal, A. and Vitter, J. S. The Input/Output Complexity of Sorting and Related Problems. *CACM*, 31(9):1116–1127, Sept. 1988.

[5] Aho, A. V. et al. *Data Structures and Algorithms*. Addison-Wesley, Reading, Massachusetts, 1983.

[6] Alpern, B. et al. Uniform Memory Hierarchies. *Algorithmica*, 12(2/3):72–109, August/September 1994.

[7] Appel, A. W. and Li, K. Virtual Memory Primitives for User Programs. In *ACM ASPLOS*, pp. 96–107, Apr. 1991.

[8] Atkinson, M. P. et al. An Approach to Persistent Programming. *The Computer Journal*, 26(4):360–365, Nov. 1983.

[9] Atkinson, M. P. and Morrison, R. Procedures as Persistent Data Objects. *ACM TOPLAS*, 7(4):539–559, Oct. 1985.

[10] Buhr, P. A. et al. μC++: Concurrency in the Object-Oriented Language C++. *SP&E*, 22(2):137–172, Feb. 1992.

[11] Buhr, P. A., Goel, A. K., and Wai, A. μDatabase : A Toolkit for Constructing Memory Mapped Databases. In *Persistent Object Systems*, pp. 166–185, Sept. 1992. Springer-Verlag.

[12] Cockshott, W. P. et al. Persistent Object Management System. *SP&E*, 14(1):49–71, 1984.

[13] Copeland, G. et al. Uniform Object Management. In *EDBT*, volume 416, pp. 253–268, Mar. 1990. Springer-Verlag.

[14] Fortune, S. and Wyllie, J. Parallelism in Random Access Machines. In *ACM STOC*, pp. 114–118, 1978.

[15] Gonnet, G. H. and Baeza-Yates, R. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1991.

[16] Gonnet, G. H. and Munro, J. I. Heaps on Heaps. *SIAM J. Comp.*, 15(4):964–971, Nov. 1986.

[17] Graefe, G. Sort-Merge-Join: An Idea whose Time has(h) Passed? In *IEEE ICDE*, page 406, February 1994.

[18] Hong, J.-W. and Kung, H. T. I/O Complexity: The Red-Blue Pebble Game. In *ACM STOC*, pp. 326–333, 1981.

[19] Johnson, N. L. and Kotz, S. *Urn Models and their Application: An approach to modern discrete probability theory.* John Wiley & Sons, 1977.

[20] Kitsuregawa, M. et al. Application of Hash to Data Base Machine and Its Architecture. *New Generation Computing*, 1(1):63–74, 1983.

[21] Lamb, C. et al. The Objectstore Database System. *CACM*, 34(10):50–63, Oct. 1991.

[22] Lieuwen, D. F. et al. Pointer-Based Join Techniques for Object-Oriented Databases. In *IEEE PDIS*, Jan. 1993.

[23] Mackert, L. and Lohman, G. Index Scans Using a Finite LRU Buffer: A Validated I/O Model. *ACM TODS*, 14(3):401–424, Sept. 1989.

[24] Martin, T. P. et al. Parallel Hash-Based Join Algorithms for a Shared-Everything Environment. *IEEE Trans. on Knowledge and Data Engineering*, 6(5):750–763, Oct. 1994.

[25] Moss, J. Working with Persistent Objects: To Swizzle or Not to Swizzle. Technical Report CS 90-38, CS Department, University of Massachusetts, May 1990.

[26] Organick, E. I. *The Multics System*. The MIT Press, Cambridge, Massachusetts, 1972.

[27] Papadimitriou, C. and Ullman, J. D. A Communication-Time Tradeoff. *SIAM J. Comp.*, 16(4):639–646, Aug. 1987.

[28] Richardson, J. E. et al. The Design of the E Programming Language. *ACM TOPLAS*, 15(3):494–534, July 1993.

[29] Schaffer, R. and Sedgewick, R. The Analysis of Heapsort. *Journal of Algorithms*, 15:76–100, 1993.

[30] Schneider, D. A. and DeWitt, D. J. A Performance Evaluation of Four Parallel Joins Algorithms in a Shared-Nothing Multiprocessor Environment. In *ACM SIGMOD*, pp. 110–121, June 1989.

[31] Shapiro, L. D. Join Processing in Database Systems with Large Main Memories. *ACM TODS*, 11(3):239–264, Sept. 1986.

[32] Shekita, E. and Zwilling, M. Cricket: A Mapped, Persistent Object Store. In *Persistent Object Systems*. Morgan Kaufmann, 1990.

[33] Shekita, E. J. and Carey, M. J. A Performance Evaluation of Pointer-Based Joins. In *ACM SIGMOD*, pp. 300–311, June 1990.

[34] Stonebraker, M. Operating system support for database management. *CACM*, 24(7):412–418, July 1981.

[35] Vitter, J. S. and Shriver, E. A. M. Algorithms for Parallel Memory I. *Algorithmica*, 12(2/3):110–147, Aug/Sep 1994.

[36] Vitter, J. S. and Shriver, E. A. M. Algorithms for Parallel Memory II. *Algorithmica*, 12(2/3):148–169, Aug/Sep 1994.

[37] White, S. J. and DeWitt, D. J. QuickStore: A High Performance Mapped Object Store. In *ACM SIGMOD*, pp. 395–406, May 1994.

[38] Wilson, P. R. Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware. *Computer Architecture News*, 19(4):6–13, June 1991.