Parallelism and its Price : A Case Study of NonStop SQL/MP

SUSANNE ENGLERT
Tandem Computers
englert_susanne@tandem.com

RAY GLASSTONE
Tandem Computers
glasstone_ray@tandem.com

WAQAR HASAN*
Stanford University
hasan@cs.stanford.edu

Abstract

We describe the use of parallel execution techniques and measure the price of parallel execution in NonStop SQL/MP, a commercial parallel database system from Tandem Computers. NonStop SQL uses intra-operator parallelism to parallelize joins, groupings and scans. Parallel execution consists of starting up several processes and communicating data between them. Our measurements show (a) Startup costs are negligible when processes are reused rather than created afresh (b) Communication costs are significant – they may exceed the costs of operators such as scan, grouping or join. We also show two counter-examples to the common intuition that parallel execution reduces response time at the expense of increased work – parallel execution may reduce work or may increase response time depending on communication costs.

All execution times reported in the paper are scaled. No inferences should be drawn about actual execution times. All query executions reported in the paper were created by bypassing the NonStop SQL optimizer. No inferences should be drawn about the behavior of the optimizer.

1 Introduction

Parallelism is now recognized as the key to providing high performance database systems at low cost [DG92, PMC+90, Val93]. While initial interest was in parallel OLTP, parallelism is now being used to enable large scale decision-support applications.

The declarative nature of the SQL language provides abundant opportunities for parallel execution. However, the use of parallelism has a price. Processing a query in parallel requires starting up several processes and communicating data among them. While linear speedup through parallelism is desirable, it is also impossible to achieve for *all* queries. For example, a scan that is localized to a single disk by selection predicates has no available parallelism. Even when parallelism is available, the speedup from parallelism may be offset by the cost of starting processes and communicating data between them.

This paper is a case study of NonStop SQL/MP, a commercial parallel database system from Tandem Computers. The case study was conducted with two goals: (a) to document the use of parallel execution techniques in a commercial system, and (b) to determine the price of using parallel execution. The second goal is in sharp contrast to the usual approach of focusing exclusively on executions where the use of parallelism is a clear win.

NonStop SQL/MP exploits intra-operator parallelism to execute joins, groupings and scans in parallel. A single table may be horizontally partitioned across multiple disks to provide IO parallelism. A single operator may be executed by multiple processes on distinct processors to provide CPU parallelism. Strategies based on data partitioning and replication are used for parallelizing operators.

We report experiments that measure the cost of starting up a query and of communicating data between processes. Startup overhead is incurred as a prelude to real work. It consists of obtaining a set of processes and passing to each a description of its role in executing the query. The description consists of the portion of the query plan the process will execute and the identities of the other processes it will communicate with.

Prior experiments [Tan] with Tandem systems have shown the rate at which messages may be passed between two processors to be limited by CPU speed rather than bus bandwidth. Thus, our experiments focus on the CPU cost of communication.

Our experiments consider three categories of communication between processes. *Local* communication consists of a producer process sending data to a consumer process on the same processor. *Remote* communication is the case when the producer and consumer are on distinct processors. *Repartitioned* communication consists of a set of producers sending data to a set of consumers. Each tuple is routed based on the value of some attribute.

Communication requires data to be moved from one physical location to another. Local communication is im-

*Supported by Hewlett-Packard Company

plemented as a memory to memory copy across address spaces. Remote communication divides data into packets which are transmitted across the inter-connect. The receiving CPU has to process interrupts generated by packet arrival as well as reassemble the data. In repartitioned communication, a producer has to perform some additional computation to determine the destination of each tuple.

Our experiments compare the cost of communication with the cost of operators such as scans, joins and groupings. We observe that while the cost of communicating data is proportional to the number of bytes transmitted, an operator may not even look at all its input data – it only needs to look at attributes that are relevant to it and may ignore the attributes that are relevant only to subsequent operators.

Our findings on the price of parallelism are:

• Startup costs are negligible when processes can be *reused* rather than created afresh.
• Communication cost consists of the CPU cost of sending and receiving messages.
• Communication costs can exceed the cost of operators such as scanning, joining or grouping.

We also present two examples of executions that run counter to the intuition that parallel execution increases total work but reduces response time. One example shows that parallelism can reduce *both* work and response time and the other shows that excessive communication costs may *increase* response time in addition to increasing work.

There are a large number of ways in which a query may be executed in parallel. Our results imply that communication costs should be accounted for in choosing among them. It would be a mis-interpretation to draw any inferences about the NonStop SQL optimizer. Our intent was to design experiments to evaluate the role of communication costs in possible parallel executions and not in the specific ones chosen by the optimizer. Thus, all our experiments were conducted by bypassing the optimizer. Specifically, the optimizer does avoid the example execution in which parallelism increases response time.

We first describe the architecture of Tandem systems in Section 2. We then discuss, in Section 3, how the declarative nature of SQL offers opportunities for parallelism. In Section 4, we describe how these opportunities are exploited by NonStop SQL/MP. We then describe our experimental results on startup costs in Section 5. Section 6 describes our results on the cost of communication. These costs are put in perspective by comparing them with costs of operators such as scans, joins and groupings. Section 7 interesting examples of parallel and sequential execution. Finally, Section 8 discusses our results and offers directions for future work.

2 Tandem Architecture: An Overview

2.1 Parallel and Fault-tolerant Hardware

Tandem systems are fault-tolerant parallel machines. For the purpose of query processing, a Tandem system may be viewed as classical shared-nothing system (see Figure 1). Each processor has local memory and exclusive control over some subset of the disks.

Processors communicate over an interconnection network. Up to 16 processors may be connected to an inter-processor bus to form a *node*. A variety of technologies and topologies are used to interconnect multiple nodes.

For fault-tolerance, each logical disk consists of a mirrored pair of physical disks. Disk controllers ensure that a write request is executed on both disks. A read request is directed to the disk that can service it faster; for example if both disks are idle, the request is directed to the one with its read head closer to the data.

We will not discuss further fault-tolerance features of the Tandem architecture since they are largely orthogonal to query processing. The interested reader is referred to [BBT88] for details.

2.2 Message Based Software

Messages implement interprocess communication as well as disk IO. Access to a disk is encapsulated by an associated set of disk processes that run on the processor that controls the disk. They implement the basic facilities for reading, writing and locking disk-resident data. An IO request is made by sending a message to a disk process. Data read by a read request is also sent back to the requester as a message. Use of a set of disk processes allows several requests to be processed concurrently. Disk processes are system processes and, for the purpose of query processing, may be regarded as being permanently in operation.

A single file may be partitioned across multiple disks by ranges of key values. This allows tables and indexes to be horizontally partitioned using range partitioning. The file system is cognizant of partitioned files and can route messages based on the key value of a requested record.

2.3 Performance Characteristics

The interconnect used for communication between processors is engineered to provide high bandwidth and performance. Experiments [Tan] have shown the message throughput between two processors to be limited by CPU speed rather than the speed of the inter-processor bus.

The programming interface for messages provides location transparency. However, the implementation mechanisms for inter and intra-processor messages are different. An intra-processor message is transmitted by a memory-to-memory copy. An inter-processor message is broken into packets and sent over the inter-connect. Packet arrival generates interrupts at the receiving CPU. The packets are then assembled and written into the memory of the receiving process. Measurements show an intra-processor message to be significantly cheaper than an inter-processor message.

A mirrored disk consists of two physical disks with identical data layout. As remarked earlier, a write request is

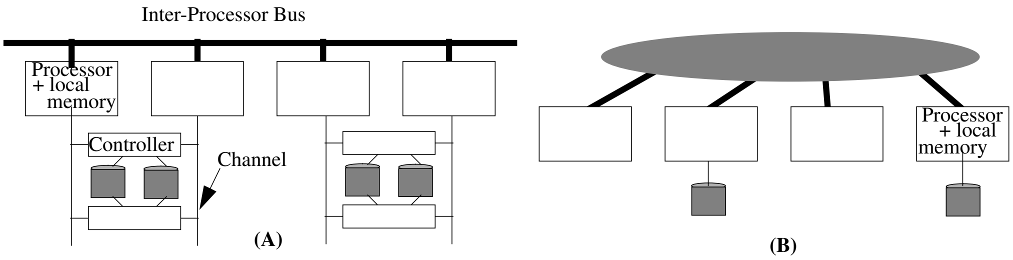


Figure 1: (a) Tandem Architecture (b) Abstraction as Shared-Nothing

executed on both physical disks while a read is directed to the disk that can process it faster. A mirrored pair processes read requests faster than a single physical disk while writes run at about the same speed.

3 Opportunities for Parallelism in SQL

In this Section, we discuss opportunities for exploiting parallelism in execution of SQL queries. The next section will discuss how such opportunities are exploited by NonStop SQL/MP.

Opportunities for parallelism fall into two categories: *intra* and *inter*-operator and have been investigated in many reseach projects [BCC+90, DGG+86, HS91, Hon92, Sch90]. Intra-operator uses several processors to execute a single operator while inter-operator executes two operators on disjoint sets of processors. We will limit our discussion to intra-operator parallelism since NonStop SQL/MP focuses on this form.

If $T = T_1 \cup T_2 \cup \ldots \cup T_k$ (where T, T_i are tables), then the unary operations of selection, projection, duplicate elimination, grouping and aggregation may be pushed through union using algebraic identities that esentially have the following form:

$$Op(T) = Op(T_1) \cup Op(T_2) \cup \ldots \cup Op(T_k)$$

The terms on the right hand side may be computed independently of each other thus providing opportunity for parallel execution. The exact transformation is more complex for operators such as grouping and aggregation.

The join of tables T and S may be parallelized by partitioning T and joining each partition with a replica of S. This strategy is called *fragment and replicate* or *partition and replicate*. The transformation applies irrespective of the nature of the join predicate; specifically it also applies to cartesian products.

$$T \bowtie S = (T_1 \bowtie S) \cup (T_2 \bowtie S) \cup \ldots \cup (T_k \bowtie S) \quad (1)$$

When an equi-join predicate is available, both T and S may be partitioned. The partitioning must guarantee that matching tuples go to matching partitions. In other words, if the value of the join column for tuple $t \in T$ matches the value of the join column for tuple $s \in S$ and t goes to partition T_i then s must go to partition S_i. The following identity shows the opportunity for partitioned parallelism.

$$T \bowtie S = (T_1 \bowtie S_1) \cup (T_2 \bowtie S_2) \cup \ldots \cup (T_k \bowtie S_k) \quad (2)$$

4 Parallelism in NonStop SQL/MP

NonStop SQL/MP uses intra-operator parallelism for scans, selection, projection, joins and grouping and aggregation. Intra-operation parallelism uses replication as well as partitioning. This Section discusses how the opportunities described in the last Section are physically exploited on the parallel architecture described in Section 2.

Inter-operator parallelism is not used. The system does not, for example, use pipelined parallelism, in which *disjoint* sets of processors are used for the producer and consumer. It does, however, use pipelined execution whenever possible, in which producers and consumers run concurrently.

In Section 4.1, we discuss the use of intra-operator parallelism. Section 4.2 discusses how operators are mapped to a processes and processes to processors.

4.1 Use of Intra-operator Parallelism

Intra-operator parallelism is based on data partitioning and replication. Recall that base tables and indexes may be stored horizontally partitioned over several disks based on key ranges. Scans and groupings are parallelized using the existing data partitioning.

Joins may repartition or replicate data in addition to using the existing data partitioning. Such repartitioning or replication occurs on the fly while processing a query and does not affect any stored data. Data repartitioning is based on hashing and equally distributes data across *all* CPUs.

Stored data is scanned by disk processes that implement selection, projection and some kinds of groupings and aggregation. Since each disk has its exclusive disk processes, the architecture naturally supports parallel scans.

Grouping is implemented in two ways, one based on sorting and the other on hashing. Sort grouping first sorts the data on the grouping columns and then computes the grouped aggregates by traversing the tuples in order. Hash grouping forms groups by building a hash table based on the grouping columns and then computes aggregates for each group.

The strategy for parallelizing a grouping is to use the existing data partitioning. A separate grouping is done for each partition followed by a combination of the results. Data is *not* repartitioned to change the degree of parallelism or the partitioning attribute.

A join of two tables (say T and S) may be parallelized in the following two ways corresponding to Equations 1 and 2 of Section 3.

Partition Both: This may be used only when an equi-join predicate is available. If both tables are similarly partitioned on the join column, the "matching" partitions may be joined. Otherwise, one or both tables may be repartitioned.

Partition and Replicate: Another parallelization strategy is to partition S and join each partition of S with all of table T. This may be achieved in two ways. The first is to replicate T on all nodes that contain a partition of S. The second is to *repartition* S (for example, to increase degree of parallelism) and replicate T on all nodes with a (new) partition of S.

Three methods are used for joins: nested-loops, sort-merge and hybrid-hash. Table 1 summarizes the join methods used for each parallelization strategy.

When both tables happen to be similarly partitioned by the join column, sort-merge join is the most efficient join method. Since the partitioning columns are always identical to the sequencing columns in NonStop SQL, the sorting step of sort-merge is skipped and the matching partitions are simply merged.

In the strategy of repartitioning both tables, both are distributed across all CPUs using a hash function on the joining columns. In this way, corresponding data from both tables or composites is located such that it can be joined locally in each CPU using the hybrid hash-join method. The strategy of repartitioning only one of the tables is not considered.

The partition and replicate strategy considers both nested-loops and hybrid-hash. The inner table is replicated and the outer table is partitioned. If the existing partitioning of the outer is used, then both nested-loops and hybrid-hash are considered. If the outer is repartitioned, then only hybrid-hash is considered.

Nested-loops join is implemented by sending a message to lookup the inner table for each tuple of the outer (thus incurring random IO in accessing the inner). The inner is replicated in the sense that if two tuples in different partitions of the outer have the same value of the join attribute, then the matching tuples of the inner will get sent to both partitions. Thus, only the relevant portion of the inner table is accessed and replication of tuples happens only if needed.

When used with partition-and-replicate parallelization, hybrid-hash join replicates the inner table. Either the existing partitioning of the outer is used or the outer is repartitioned across all CPUs. A hash table is built on the inner at each participating CPU and subsequently probed by tuples from the inner. When used with partition-both parallelization, both tables are repartitioned across all CPUs. The hybrid-hash join algorithm has adaptive aspects such as adjusting to the amount of available memory. The interested reader is referred to Zeller and Gray [ZG90] for details.

Nested-loops accesses only the relevant tuples of the inner table. Since hybrid-hash accesses the entire inner, it avoids the random IO incurred by nested-loops but also accesses tuples of the inner that may not join. Nested-loops is the only applicable method when there is no equi-join predicate.

4.2 Process Structure

A single SQL query is executed by use of multiple processes. Three kinds of processes are used. First, there is the SQL *Executor* process which consists of system library routines bound into the user application. Second, slave processes called ESPs (for Executor Server Process) may be spawned by the Executor. Third, there are disk processes which are system processes that are permanently in operation.

Scans are implemented by disk processes and the remaining work is divided between ESPs and the Executor. The query result is produced by the Executor. The mapping of operators to processes and allocation of processes to processors may be understood with respect to query trees in which interior nodes represent operations such as joins and groupings and leaves represent scans. The basic idea in forming processes is to have an operator share processes with the prior (child) operator as far as possible. New processes are created only when such combination is impossible due to a data repartitioning or due to the fact that the prior operator is a scan. In the case of a join there are two children. Since once of them is always a base table or index, the join is attempted to be combined with the operator that produces the outer table.

Scans (the leaves of a query tree) are always executed by disk processes. Thus scans are parallelized based on the partitioning of the data being read; there is one process for each disk that contains a partition of the data. While ESPs are capable of repartitioning their output, disk processes are not. Thus if the result of a scan is to be repartitioned, one ESP is created per existing partition of the data for the sole purpose of repartitioning data.

A grouping is always parallelized based on the existing partitioning of the data. It can be combined into the same process as the prior operator unless the prior operator is a scan and the grouping is such that a disk process cannot implement it. Disk processes can implement groupings in which the grouping columns are a prefix of the key columns.

The process structure for joins is more complex since a join has two operands. One of the operands, the inner, is always a base table. For nested-loops and merge-join, one ESP is used per partition of the outer table. If possible, this ESP is the same ESP as for the operator that produces the outer table. The inner is accessed by sending messages to disk processes. In the case of nested-loops, one message is sent per tuple of the outer so as to retrieve only the relevant tuples.

We only describe the process structure of hybrid-hash

	Partition Both		*Partition and Replicate*	
	Use Existing Partitioning	Repartition both	Existing Partitioning for one replicate other	Repartition one replicate other
hybrid-hash	×	√	√	√
nested-loops	×	×	√	×
Sort-merge	√	×	×	×

Table 1: Parallelization Strategies and Join Methods

for the case when both operands are repartitioned. One ESP is used per existing partition of the inner to repartition data. If the outer is a base table, one new ESP is used per partition of the outer to repartition data. On the other hand, if the outer is not a base table, then the ESP that produces it also performs the repartitioning. One ESP is used at each CPU to receive the repartitioned data and locally compute a hybrid-hash join.

5 Startup Costs

Parallel execution requires starting up a set of processes and communicating data among them. This section measures startup cost and the next section focuses on communication.

When a query is executed in parallel, the Executor process starts up all necessary ESP processes and sends to each the portion of the plan it needs to execute and the identities of the other processes it needs to communicate with. The ESP processes are created sequentially; each process is created and given its plan before the next process is created. ESPs are not terminated for 5 minutes after the query completes. In case another query is executed within five minutes, ESP processes are reused.

We measured the cost of starting up processes by running a query that required 44 ESP processes. Figure 2 plots the time at which successive processes got started and had received their portion of the plan. The dotted line plots process startup when new processes had to be created. The solid line plots the case when processes were reused.

We conclude that communicating the relevant portion of the plan to each ESP has negligible cost. Startup cost is negligible when processes can be reused. Startup incurs an overhead of 0.5 sec per process that needs to be created. A possible enhancement would be to start the ESP processes in parallel instead of sequentially.

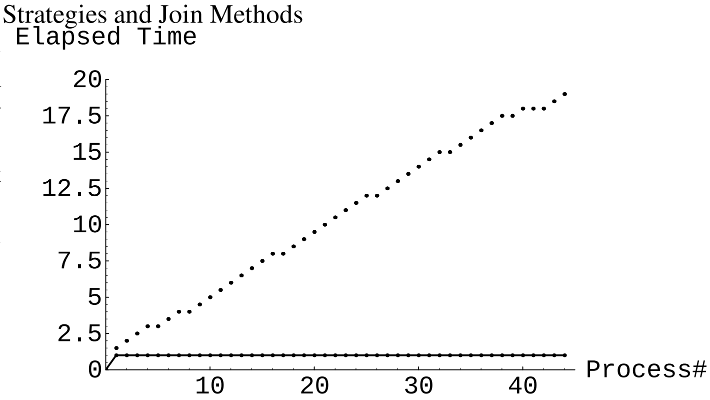


Figure 2: Process Startup: With (solid) and without (dotted) process reuse.

6 Costs of Operators and Communication

In this section we measure the cost of communication and put these costs in perspective by a comparison with operators such as scans, joins and grouping.

We describe measurements of the cost of local, remote and repartitioned communication. Local communication consists of a producer process sending data to a consumer process on the same processor. Remote communication is the case when the producer and consumer are on distinct processors. In repartitioned communication, a set of producers send data to a set of producers. The cost of repartitioning varies with the pattern of communication used. We decided to focus on the case where a *single* producer partitions its output *equally* among a set of consumers. This simple pattern captures the overhead of a producer sending data to multiple consumers i.e. the additional overhead of determining the the destination of each tuple. The producer applies a hash function to an attribute value to determine the CPU to which the tuple is to be sent. Figure 3 illustrates the forms of communication covered by our experiments. These cases were chosen due to their simplicity. The costs of other communication patterns may be extrapolated.

Table 2 summarizes the results of our measurements. It turned out that the cpu time of all our queries was linear in the amount of data accessed. Even operations that involved sorting behaved linearly in the range covered by our experiments. Thus costs are stated in units of msec/Ktuple and msec/Mbyte. The two units are comparable since 1K tuples occupy 1 MByte for the table under consideration. Join costs were measured by joining two tables, each with k tuples, to produce k output tuples. Join costs were linear in k and are therefore reported in msec/Ktuples.

Our approach was to devise experiments such that the cost of an operation could be determined as the difference of two executions. For instance the cost of local communication was determined as the difference of executing the same query using two plans that only differed in whether one or two processes were used.

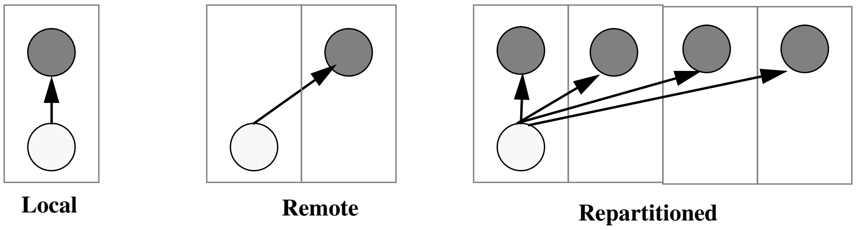


Figure 3: Local, Remote and Repartitioned Communication

Transfer Operation	*Cost (msec/Mbyte)*	*Computational Operation*	*Cost (msec/Ktuple)*
Scan	180	Aggregation	65
Local Comm.	390	Sort-Merge Join	370
Remote Comm.	745	Hash Join	40
Repartitioning (4 CPUs)	1230	Hash Grouping	110
		Sort Grouping	765

Table 2: CPU Costs of Operations (1K tuples occupy 1 MByte)

Section 6.1 provides an overview of our experimental setup. Sections 6.3 and 6.4 describe experiments that measure the cost of communication and Sections 6.2, 6.5 and 6.6 address the costs of operators.

6.1 Experimental Setup

We ran all experiments reported in this Section on a 4 processor Himalaya K1000 System. Each processor was a MIPS R3000 processor with 64MB of main memory and several 2 GB disks. The size of the cache associated with each disk was reduced to 300 Kbytes to reduce the effects of cacheing on our experiments.

The tables `Single`, `Single2` and `Quad` used in our experiments had identical schema and content. `Quad` was equally partitioned over four disks while `Single` and `Single2` were stored on single disks.

Each of these tables had four columns: `unique`, `twenty`, `hundred` and `str`. The first three were integer columns and the fourth a 988 byte string. The `unique` column was the key and each table was stored sorted by this column. The column `twenty` was randomly chosen from $1 \ldots 20$, `hundred` randomly chosen from $1 \ldots 100$, and `str` was a 988 byte string with an identical value in each row. Each tuple occupied 1000 bytes. Each table had 50,000 tuples resulting in a total size of 50 MBytes.

We forced query plans by the use of optimizer hooks that allowed us to specify plan elements such as the sequence and method for each join; whether parallel execution should be used or not; and whether a join should repartition data or not, whether predicates should be combined with a disk process or not and so on. The EXPLAIN command in NonStop SQL allowed us to view plans to confirm the details of the execution.

We collected performance data by using MEASURE, a low overhead tool. MEASURE collects statistics about objects of interest such as processors, disks, processes and files while a program is in execution. The collected statistics can later be perused using a query tool. MEASURE also measures the cost of processing interrupts that are generated by message arrival and IO completions – these costs are not assigned to any process.

Each data point reported in this paper is an average over three executions. Typically, the three executions differed by less than 1%. All plotted curves were obtained using a least squares fit using the Fit function in Mathematica.

6.2 Costs of Scans, Predicates and Aggregation

We used the following query to scan `Single`.

Query1: **select** unique **from** Single
where twenty $>$ 50000 **and** unique $< k$

The predicate `twenty >50000` is false for all tuples. Thus no tuples are returned and the overhead of communicating the result of the scan is eliminated. Since the table was stored sorted by `unique`, the predicate `unique` $< k$ allowed us to vary the portion of the table scanned.

The query plan used a single disk process and combined predicate evaluation with the scan. The cost of the plan consists of a scan and two predicate evaluations, one of which is a key predicate. The dotted line in Figure 4 plots the cost as k was varied from 5000 to 50000 in increments of 5000. Denoting cpu cost by t and the number of Mbytes scanned by b, a least squares fit yields the equation $t = 0.31 + 0.185b$. Thus a scan with two predicates costs 185 msec/MByte.

We determined the cost of predicate checking by additional measurements. To measure the cost of the key predicate, we tried two queries: one with the predicate `unique` $<$ `100,000` and the other with no key predicate. Both queries scanned the entire table since all key values were less than 100,000 and ran in identical time.

To measure the cost of the non-key predicate, we ran a query with *two* non-key predicates. The "where clause" of Query1 was changed to `(twenty >50000` **or** `hundred >50000)` **and** `unique1` $< k$. The solid lines in Figure 4 plots the cost of a query. Curve fitting yields $t = 0.31 + 0.18b$ i.e. the cost increases by 5 msec/MByte due to the additional non-key predicate. Thus, we may expect a scan with *no* predicates to cost 180 msec/MByte.

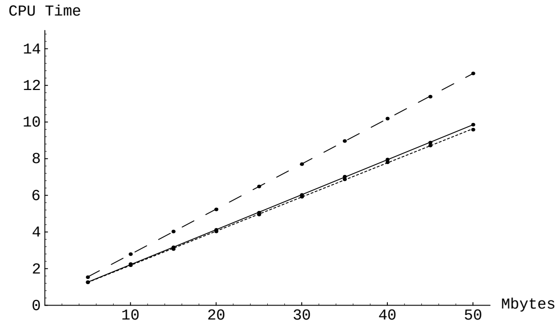


Figure 4: Scan with 1 predicate(dotted), 2 predicates(solid), aggregation(dashed)

The dashed line in Figure 4 shows the cost of applying an aggregation in the disk process using the following query.

Query2: **select** max(str) **from** Single
where unique $< k$

A least square fit yielded the equation $t = 0.31 + 0.245b$. Subtracting scan cost, we infer aggregation to cost (245-180) msec/MByte which is 65 msec/MByte. Recall that `str` is a 988 byte string with an identical value in each row. Thus the aggregation uses 988 bytes of each 1000 byte tuple.

6.3 Costs of Local and Remote Communication

We measured the cost of local and remote communication by use of optimizer hooks that permitted the creation of plans in which the aggregation in Query2 was moved to a separate process (the Executor) and the process could either be placed on the same CPU as the disk process or on a different CPU. Figure 6 shows the process structure for the three executions.

When aggregation is in a separate process from scan, 988 bytes of each 1000 byte tuple have to be communicated across processes. Figure 5(a) plots the data points for scanning and aggregation in the disk process and also with the remote and local communication. The curves are marked (a), (b) and (c) to show the correspondence with Figure 6. Least squares curve fitting shows slopes of 0.635 and 0.99 for the local and remote curves. Since scanning and aggregation without communication has a slope of 0.245, we infer that local communication costs 390 msec/Mbyte and remote communication costs 745 msec/MByte.

We observe that the relative cost of communication is a function of the amount of data communicated. Figure 5(b) shows the case when Query2 is modified to aggregate on `twenty`. In this case only 4 bytes of each 1000 byte tuple have to be communicated across processes and the relative cost of communication is negligible.

6.4 Cost of Repartitioned Communication

Repartitioning dynamically distributes data across all CPUs using a hash function. In general this involves a combination of local and remote communication. Since tuples are routed based on a hash function applied to some column, additional cost of deciding the destination must be incurred for each tuple.

Given a system with 4 CPUs, we chose to focus on the case where a single producer equally repartitions data among four consumers. Since one consumer was placed on the same CPU as the producer, 1/4'th of the tuples may be expected to be transported using local messages and the remaining 3/4'th by remote messages. The cost of repartitioning will vary depending on the number of CPUs and the arrangement of producers and consumers.

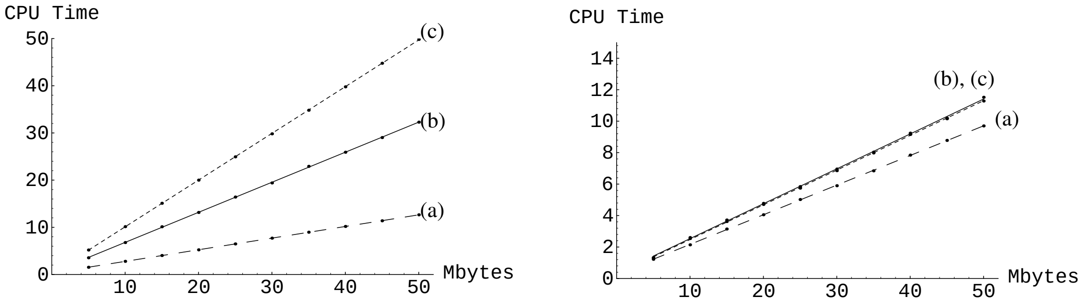
We devised the following query to create two executions that only differ in whether or not data is repartitioned. `Small` is a single column table with twenty values 0..19 stored in twenty tuples. The result of joining `Single` and `Small` is identical to `Single` and is grouped into twenty groups.

Query3: **select max**(str) **from** Single w, Small s
where w.twenty = s.unique and w.unique $< k$
group by w.twenty

We forced the two executions shown in Figure 7. Both use a simple hash join in which a hash table is built on `Small` and probed by `Single`. This is followed by a hash grouping. The first execution executes the join and grouping in the Executor process on a single CPU. The second execution build a hash table on `Small` and replicates it on four CPUs. Then `Single` is repartitioned and the join and grouping computed separately for each partition. Finally, the Executor process merges the results of the separate groupings.

While Figure 7(b) shows several extra communication arrows, only the repartitioning arrows are significant. Between 5 and 50 Mbytes of data is repartitioned. In comparison, the hash table on `Small` occupies about 0.00008 Mbytes and thus replicating it has negligible cost. The result of each grouping consists of 20 groups which is about 0.02 Mbytes, which is also negligible in comparison.

Figure 8 plots the costs of the two executions as as k was varied from 5000 to 50000 in increments of 5000. Least squares curve fitting shows the slopes of the lines to be 0.785 and 2.015. Since the difference between the two executions is the cost of repartitioning, we conclude repartitioning to cost $(2.015 - 0.785)$ sec/MByte or 1230 msec/MByte. We remind the reader that our measurements of repartitioning cost are for four CPUs.



(A) 98.8% of scanned data communicated (B) 4% of scanned data communicated

Figure 5: Scan and Aggregation(dashed) with Local(solid) and Remote(dotted) Comm.

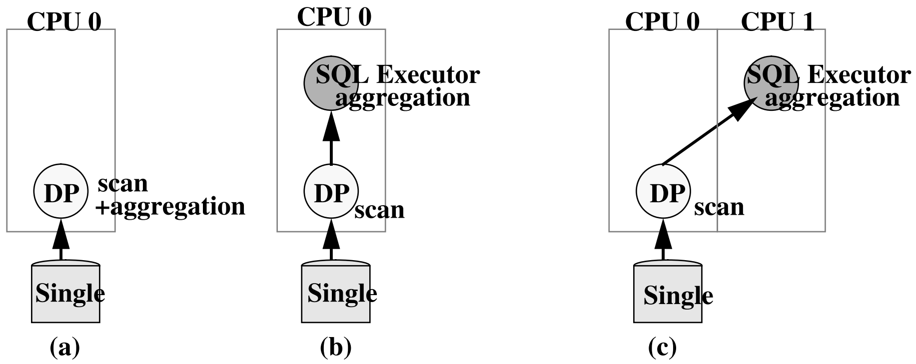


Figure 6: Process structure: (a) no communication (b) Local (c) Remote

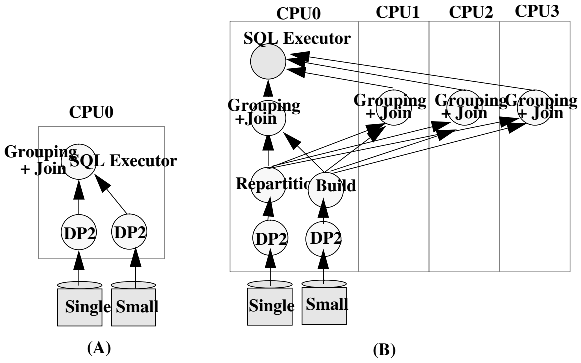


Figure 7: Local and Repartitioned Execution

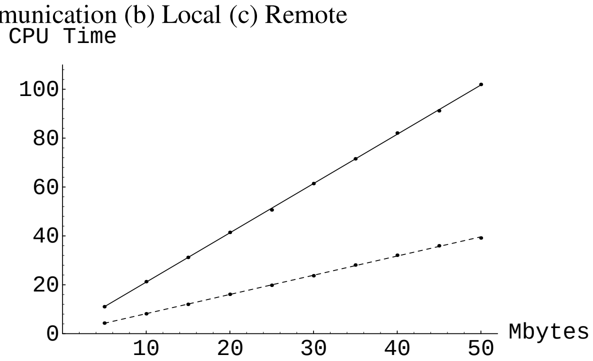


Figure 8: Local(dotted) and Repartitioned(solid) Comm.

6.5 Costs of Join Operators

We measured the cost of simple-hash, sort-merge and nested joins by joining `Single` with an identical copy called `Single2`. We executed the following query using different join methods. The query was modified for sort-merge join to require sorting on one operand by changing the join predicate to `w1.unique = w2.hundred`. Figure 9 plots the execution costs as k was varied from 5000 to 50000 in increments of 5000.

Query4: **Select max**(w1.str) **from** Single w1, Single2 w2 **where** w1.unique = w2.unique **and** w1.unique $< k$ **and** w2.unique $< k$

Surprisingly *all* plots in Figure 9 are linear in k even though we are joining two operands each with k tuples, and producing a result consisting of k tuples.

The nested join accesses the inner table (`Single2`) for each tuple of the outer (`Single`). Thus the cost is linear in the size of the outer table. Each access to the inner table is a random IO which explains the high cost of the nested join.

Hash-join builds a hash table on the qualifying tuples of `Single2` and probes it using tuples from `Single`. The one possible source of non-linearity is when k probes are performed on a hash table that contains k entries. We

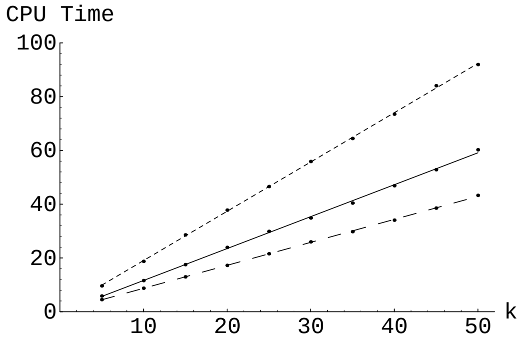


Figure 9: Query using Simple-hash(dashed), Sort-merge(solid) and Nested Join(dotted)

conclude that cost of a probe is independent of hash table size.

For sort-merge join, only one operand (`Single2`) needed to be sorted since the other was pre-sorted on the join column. It may be surprising that the cost of sorting does not introduce any non-linear component into the cost. The explanation is that the system chose to sort by inserting tuples into a sequenced file. The cost of insertion is independent of file size and the cost of comparisons is not a significant cost in locating the correct page.

Least squares curve fitting shows cost of the query to be 1835, 855 and 1185 msec/Mbyte for nested, hash and sort-merge join respectively. The "per Mbyte" should be interpreted as "per Mbyte of each operand".

We may separate the cost of joining from the cost of scans, communication, and aggregation by using our prior measurements.

For hash-join, we incur a scan for each operand. However, local communication is significant only for `Single`. After projection, `Single2` is reduced to 4/1000'th of its original size while almost all (992/1000'th) of `Single` is communicated. Thus the cost of the join may be calculated by subtracting the cost of two scans, the cost of locally communicating `Single`, and the cost of aggregation. This gives us $855 - (2 * 180 + 390 + 65) = 105$ msec/MByte.

Similarly, the cost of a sort-merge join may be calculated to be 370 msec/MByte. The cost of a nested-loops join cannot be broken down in this manner since it incurs a random IO per tuple of `Single`.

6.6 Costs of Grouping Operators

NonStop SQL uses two algorithms for grouping. Hash grouping forms groups by hashing tuples into a hash table based on the value of the grouping column. Sort grouping forms groups by sorting the table on the grouping column. The following query reads k records and forms twenty groups.

Query5: **select max**(str) **from** Single
where unique $< k$
group by twenty;

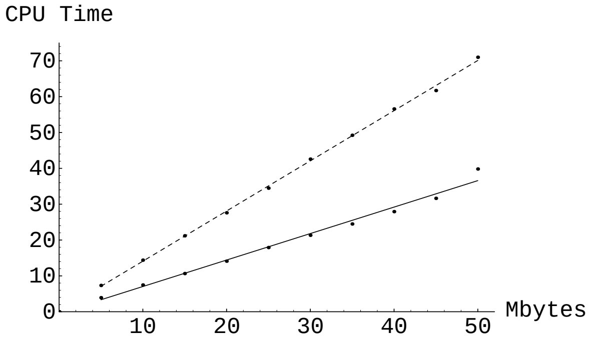


Figure 10: Hash(solid) and Sort(dotted) Grouping Costs

Figure 10 plots the costs of hash and sort grouping as a function of k. Least squares curve fitting shows the query to cost 1245 msec/Mbyte and 1400 msec/Mbyte respectively for hash and sort grouping. Since the query incurs a scan, local communication, and aggregation, we conclude that hash and sort grouping to cost 110 msec/Mbyte and 765 msec/Mbyte respectively.

7 Parallel Versus Sequential Execution

The distinction between parallel and sequential execution in Tandem systems is the use of multiple versus single *SQL Executor* processes to execute a query. Note that sequential execution may use multiple disk processes if it accesses data from multiple disks.

Parallel and sequential execution may be compared based on two metrics: work and response time. The common intuition is that parallel execution reduces response time at the expense of increased work. The basis for this intution is that parallel execution will cost at least much as sequential execution and will run at least as fast as sequential execution. While true in some cases, this is *not true* in general. The relative costs of parallel and sequential execution depend on communication costs.

We present two examples in this section. The first shows that parallel execution can *reduce* both work and response time by saving communication costs. The second shows that parallel execution can result in *increased* response time when the communication costs offset the benefit from parallel execution. We are not aware of any instances of the remaining logical possibility of parallel execution offering reduced work but increased response time compared to sequential execution.

To sum up, in addition to the intuitive case in which parallel execution runs faster but consumes more resources, it is possible that (a) parallel execution consumes less resources as well as runs faster and (b) parallel execution consumes more resources as well as runs slower. The main determinant is the cost of communication.

7.1 Parallelism can Reduce Work

The following query performs a grouping on a table that is equally partitioned across 4 disks, each attached to a distinct

CPU.

Query6: **select max**(str)
from Quad
group by twenty;

Figure 11 shows the process structure for sequential and parallel execution. When sequential execution is used, SQL runs as a single process (Executor). This process must incur remote communication to read the three partitions that reside on remote disks. When parallel execution is used, the grouping is partitioned. Each partition of `Quad` is grouped separately by an ESP process. The result of each grouping is communicated to the Executor to produce the combined grouping. The local grouping at each CPU substantially reduces the amount of data to be communicated. This results in reduced work. Response time is reduced both because of work reduction as well as better load balancing.

When sequential execution was used the query used 49 sec CPU and had a response time of 78 sec. With parallel execution, the total CPU time fell to 36.5 sec and the response time fell to 26.5 sec.

7.2 Parallelism can Increase Response Time

Consider the query used in Section 6.4 with the sequential and parallel executions shown in Figure 7. The parallel execution incurs greater work due to communication costs. Its response time is also increased since the parallelism available in the plan does not suffice to offset the increased work.

Consider the data point for $k = 50000$. When sequential execution was used the query used 39 sec CPU and had a response time of 66.5 sec. With parallel execution, the total CPU time rose to 102 sec and the response time rose to 109.5 sec.

Surprisingly, the response time increases to 109.5 sec even though 102/4 is less than 39. The explanation lies in the fact that there are sequential portions of the query and the benefit from parallelism is offset by communication costs for the parallel portions. Scanning and repartitioning `Single` is inherently sequential. These operations can only be performed on CPU 0. Parallel execution only benefits the join and grouping. That speedup is not sufficient to offset the increase in work due to repartitioning. No parallelism is available in scanning `Small` and building and replicating a hash table on it. However, these operations had negligible cost compared to the rest of the query.

It should be noted that the inherent sequentiality illustrated in this example is not pathological. Selection predicates can localize a scan to a single disk (or a subset of the disks) even when a table is partitioned across several disks.

8 Conclusions and Future Work

Our findings on the price of parallelism in NonStop SQL/MP may be summarized as:

- Startup costs are negligible when processes can be *reused* rather than created afresh.
- Communication cost consists of the CPU cost of sending and receiving messages.
- Communication costs can exceed the cost of operations such as scans, joins or grouping.

Our experiments show that the cost of parallel execution can differ substantially from that of sequential execution. The cost may be more or even less depending on what data needs to be communicated.

The cost of communication relative to the cost of operators is a strong function of the quality of the implementation. For example if operators are poorly implemented, communication costs will be relatively low. Further, such a poor implementation may actually lead to the system exhibiting good scalability! This underlines the fact that scalability must be tested with respect to the *best* implementation on a *uni-processor*.

An interesting question is how communication can be avoided or its cost reduced. Architectural techniques such as DMA are likely to help to some extent. However, most of the cost of communications tends to be incurred at software levels that are higher than DMA interfaces. Use of shared-memory is of limited value since the cost of communication through a shared piece of memory rises as the number of processors increases.

We believe that communication costs can be reduced by explicit incorporation into the algorithms used for data placement, query processing and optimization.

Data placement techniques can avoid or reduce communication. Allocating horizontal partitions so as to co-locate partitions of different tables that are frequently joined avoids repartitioning. Vertical partitioning can save communication by delaying the retrieval of a column until a later stage in the query plan. Replicas with different data partitioning can increase the chances of avoiding communication. At the query processing level techniques such as semi-joins and data compression may reduce communication costs.

Though research in distributed query optimization [CP84, OV91, YC84] focussed on communication costs, it used models of execution that did not incorporate cognizance of parallel execution. Most work in parallel query optimization [SE93, SYT93, CLYY92, HLY93, ZZBS93, GHK92] has ignored communication costs. Parallel query optimization techniques that work with cost models that include communication costs are needed [Has95, HM94, HM95, CHM95]. Further, optimization algorithms that can exploit horizontal and vertical

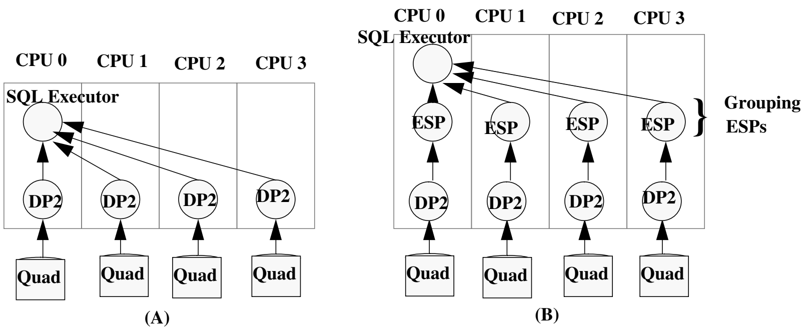


Figure 11: Process Structure: Sequential and Parallel Execution

partitioning as well as replication are an interesting direction.

Acknowledgements: We thank Harry Leslie and Gio Wiederhold for useful discussions.

References

[BBT88] B.Ball, W. Bartlett, and S. Thompson. Tandem's Approach to Fault Tolerance. *Tandem Systems Review*, 4(1), February 1988. Part Number 11078.

[BCC+90] H. Boral, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, A Highly Parallel Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[CHM95] C. Chekuri, W. Hasan, and R. Motwani. Scheduling Problems in Parallel Query Optimization. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1995.

[CLYY92] M-S Chen, M-L Lo, P.S. Yu, and H.C. Young. Using Segmented Right-Deep Trees for the Execution of Pipelined Hash Joins. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 15–26, June 1992.

[CP84] S. Ceri. and G. Pelagatti. *Distributed Database Design: Principles and Systems*. McGraw-Hill, 1984.

[DG92] D. J. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[DGG+86] D. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna. GAMMA: A High Performance Dataflow Database Machine. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, August 1986.

[GHK92] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query Optimization for Parallel Execution. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 9–18, June 1992.

[Has95] W. Hasan. *Optimization of SQL Queries for Parallel Machines*. PhD thesis, Stanford University, 1995. In preparation.

[HLY93] K.A. Hua, Y. Lo, and H.C. Young. Including the Load Balancing Issue in The Optimization of Multi-way Join Queries for Shared-Nothing Database Computer. In *Second International Conference on Parallel and Distributed Information Systems*, San Diego, California, January 1993.

[HM94] W. Hasan and R. Motwani. Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelined Parallelism. In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 36–47, Santiago, Chile, September 1994.

[HM95] W. Hasan and R. Motwani. Coloring Away Communication in Parallel Query Optimization. In *Proceedings of the Twenty First International Conference on Very Large Data Bases*, Zurich, Switzerland, September 1995. To Appear.

[Hon92] W. Hong. *Parallel Query Processing Using Shared Memory Multiprocessors and Disk Arrays*. PhD thesis, University of California, Berkeley, August 1992.

[HS91] W. Hong and M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, December 1991.

[OV91] M.T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, 1991.

[PMC+90] H. Pirahesh, C. Mohan, J. Cheung, T.S. Liu, and P. Selinger. Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches. Technical report, IBM Research Division, September 1990. IBM Research Report RJ 7724.

[Sch90] D. A. Schneider. *Complex Query Processing in Multiprocessor Database Machines*. PhD thesis, University of Wisconsin—Madison, September 1990. Computer Sciences Technical Report 965.

[SE93] J. Srivastava and G. Elsesser. Optimizing Multi-Join Queries in Parallel Relational Databases. In *Second International Conference on Parallel and Distributed Information Systems*, San Diego, California, January 1993.

[SYT93] E. J. Shekita, H.C. Young, and K-L Tan. Multi-Join Optimization for Symmetric Multiprocessors. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, Dublin, Ireland, 1993.

[Tan] Tandem. Cyclone/R Message System Performance. Technical report, Tandem Computers.

[Val93] P. Valduriez. Parallel Database Systems: Open Problems and New Issues. *Distributed and Parallel Databases: An International Journal*, 1(2):137–165, April 1993.

[YC84] C.T. Yu and C.C. Chang. Distributed Query Processing. *ACM Computing Surveys*, 16(4), December 1984.

[ZG90] H. Zeller and J. Gray. Hash Join Algorithms in a Multiuser Environment. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, Brisbane, Australia, 1990.

[ZZBS93] M. Ziane, M. Zait, and P. Borla-Salamet. Parallel Query Processing in DBS3. In *Second International Conference on Parallel and Distributed Information Systems*, San Diego, California, January 1993.