SCALABILITY OF PARALLEL JOINS ON HIGH PERFORMANCE MULTICOMPUTERS

A. G. Dale,* F. F. Haddix,**
R. M. Jenevein,* and C. B. Walton*

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-89-17 June 1989

ABSTRACT

This paper focuses on parallel joins computed on a mesh-connected multicomputer. We propose a cost-effective sort engine and a novel algorithm that combines hashing and semijoins. An analytic model is used to select hardware configurations for detailed evaluation and to suggest refinements to the algorithm. Simulation of our model confirmed the analytic results. Results indicate that parallel joins scale very well. In some cases, synergistic effects lead to better than linear speedup.

Keywords: scalability, parallelism, interconnection networks, wormhole routing, hashing, semijoins, relational databases

* Department of Computer Sciences.
** Applied Research Laboratories.

1 Introduction

We are currently engaged in research investigating low-level relational database processing in parallel machine architectures[1]. In particular, we have investigated problems of effectively performing join operations on data distributed over multiple storage nodes. Effective exploitation of parallelism for this basic operation is a key to high-performance processing of relational databases. The research discussed in this paper focuses on the performance of join operations mapped to a mesh-connected multicomputer[2]. In this paper, we examine the performance of a particular distributed join algorithm over a range of configurations. On the basis of these performance studies we intend to implement a relational database management kernel on a specific configuration to explore the use of general purpose multicomputers for high performance database processing.

2 Configuration and Data Partitioning

2.1 Background

Our previous research has concentrated on the Kyklos architecture [Mene85]. The global architecture of Kyklos includes the following major components:

1. A binary tree-based interconnection network. The second and subsequent trees are shuffled so that each tree has a unique connection permutation to the common leaves.

2. A node mapper [Desa88], which is a specialized hashing processor used to map relations to storage nodes. Conceptually, the node mapper(s) is located at the roots of the binary trees.

3. Processing nodes (P-nodes) located at the interior nodes of the trees, consisting of a general purpose microprocessor and network connections.

[1]This research was partially funded by the Applied Research Laboratories Independent Research and Development Program.

[2]The system available to us is funded by a Facilites Improvement Grant from the Shell Company Foundation to the Department of Computer Sciences, The University of Texas at Austin.

4. I/O nodes located at the leaves of the trees. Each I/O node has one connection to each tree. Major subcomponents of the I/O node include the following:

    - Processing element used to coordinate node activities and to handle external and within-node communications.
    - Sort /search/set engine [Bhat87] based on board-level integration of CADM (Content Addressable Data Manager) chips developed by Advanced Micro Devices and a Motorola MC68020 microprocessor.
    - Mass Storage Device (MSD), including microprocessor, associative disk cache, CADM disk index and conventional moving head disks.

The need for a platform to use in emulating Kyklos, our interest in the potential effectiveness of "wormhole routing" schemes and the need for a VME bus interface for each CPU turned us to the Symult 2010 architecture. "Wormhole routing" [Dall86] is a term coined to describe a communication scheme that exploits specialized communications processors in the interconnection network to avoid main processor interrupts. Although much of the Kyklos research assumes a "store and forward" communication scheme, it is, nevertheless, applicable in the new environment.

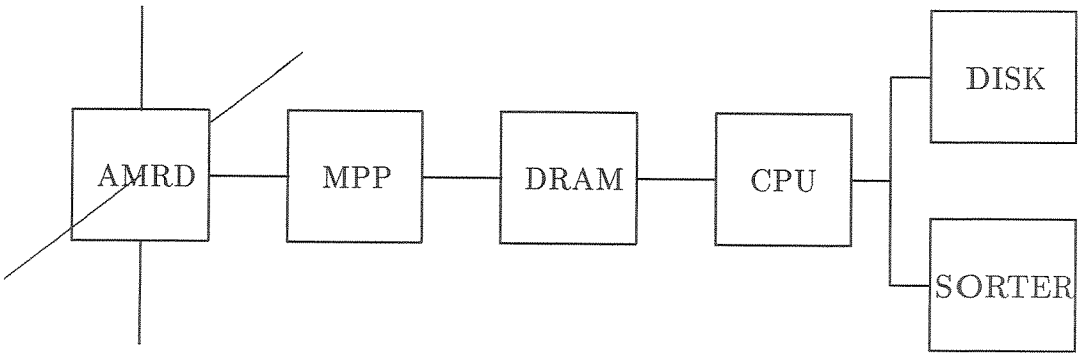
2.2 Configuration

The Symult 2010 [Symu89] is a mesh-connected multicomputer. The current configuration includes computational nodes with and without disks and is hosted by a Sun workstation. The Reactive Kernel operating system, a variant of UNIX, is resident on each computational node and provides memory managment, message dispatching and receiving, process management and file access.

Figure 1 illustrates a proposed node architecture, which is considered in this paper. It includes the following major components:

1. Automatic message routing device (AMRD). Message traffic and routing within the network is handled by the AMRD's without making any demands on the resources of intermediate nodes along the path from source to destination.

Figure 1: Proposed Single Node Structure



2. Message packet processor (MPP). The message packet processor transfers message contents directly into main memory. This means that the CPU is not interrupted by receipt of a message, while allowing the interconnection network to be cleared for other traffic.

3. The main memory ranges from 1 to 8 megabytes of 32-bit dynamic random access memory (DRAM).

4. The central processing unit (CPU) contains a MC68020 microprocessor unit running at 25 MHz, MC68881 floating point coprocessor, and memory management unit.

5. One or more conventional moving head disks (DISK).

6. Sort Device (SORTER), including Content Addressable Data Manager chips, and utilizing the CPU as a sort engine controller.

Note that in configuring a system, addition of MSD[3] and SORTER[4] to a basic processing node is optional. Components 1–5 above are available from Symult.

[3]The Mass Storage Driver envisioned in the Kyklos architecture is not used here.

[4]A prototype of the sort engine is now under development, and we intend to incorporate this device into the node structure of the Symult 2010. The performance studies in this paper assume that this sort engine is included in each of the sorting nodes and utilize the timing data developed in [Bhat87].

2.3 Data Partitioning

No tuple clustering is assumed in this study, and no secondary indexes are assumed. We do assume that the relations that comprise the database are horizontally partitioned across the disks, with equal-size partitions, utilizing the node mapper described above. In processing the join, the results are delivered to the host machine, sorted on a key specified by the host, not related to the join attributes. The overall join strategy used herein was developed for Kyklos; the principal difference between the two instantiations is in the heuristic for join site determination.

3 Parallel Join Algorithm

3.1 Environment

The parallel join algorithm is designed to operate in an environment such as that described in Section 2. The proposed sort engine and wormhole routing make the use of a sort-intensive distributed algorithm practiceable.

This algorithm considers the possiblity of software pipelining between heterogeneous processing elements. Subject to the limitations described below, if D is the order of magnitude of the relational tuple cardinalities and P is the order of magnitude of processor cardinalities, join performance of $O(D * (logP + logD))$ is claimed. If relation cardinalities are smaller than optimal for the network and algorithm, then only part of the network (as allocated by the distribution algorithm) is used.

A modified node mapper distribution algorithm [Desa88] is used. N is the subset of disk nodes on which fragments of the large relation R are stored, and M is the number of disk nodes on which fragments of the small relation S are stored ($M \leq N$). The above-referenced node mapper algorithm has been modified so that M and N are powers of 2, $\leq P$.

3.2 Logical to Physical Mapping

Physical nodes consist of a CPU, main memory, connections to other nodes (if more than one node exists) and optionally, peripherals, such as conventional moving head disks and specialized sort devices. One or more logical nodes may be mapped to one physical node. Three types of logical nodes are

considered: sort nodes are assumed to have a specialized sort capability; disk nodes are assumed to have disks; merge nodes are assumed to be general purpose processors only, used for performing merge-joins. The number of disk nodes is assumed to be $N = 2^n$, where n is a non-negative integer. The number of merge nodes is assumed to be $K_m N$, where $K_m = 2^m$, where $-2 < m < 2$ and m is an integer. The number of sort nodes is assumed to $K_s N$, where $K_s = 2^k$, where $-2 < k < 0$ and k is an integer.

A logical cluster has at least one of each type of logical processor. Cluster is used here to mean the smallest partition of a machine into homogeneous fragments, i.e., the smallest division such that each fragment contains an identical set of components. If no such partition of a machine is possible, it is composed of only one cluster. If the number of clusters is N/K_c, where $K_c = 2^c$, $0 \leq c \leq 2$ and c is an integer, there are K_c disk nodes, $K_m K_c$ merge nodes and $K_s K_c$ sort nodes per cluster. Within a cluster, the three logical processor types may be mapped to one, two or three physical processor types. If multiple functions are mapped to a single physical processor, intra-cluster data communications are reduced.

3.3 Limitations

Within a limit of $(Size\_of\_CADM/(sort\_key+pointer\_size+tag\_size))^2$, linear sort performance is achievable with a sort engine using a modified off-line sort algorithm [Bhat87]. For larger cardinalities, performance asymptotically approaches $NlogN$.

Scalability is also limited by communications. The amount of data transmitted is linear; however, the average distance over which the data is transmitted is $O(N^{3/2})$, due to the mesh topology used herein. Although performance may be enhanced by use of a node mapper, attribute distribution histograms or attribute based storage schemas, the absence of a correlation between location of tuples between the two relations makes reducing the order of average distances impossible. Within the instant topology, the distance traversed element of communications cost is $O(10^{-7})$, which implies that the $O(N^{3/2})$ average distance[5] does not significantly influence performance for networks where $N \leq 2^{12}$. The comparison of the effects of increasing net-

[5]This is a limitation of the toplogy, not the technology [Flai87], which allows higher dimensions and slower growth in average distance.

work size on time per message for wormhole and store-and-forward routing, shown in Table 1, illustrates this.

Table 1: Expected Transmission Times (μs) by Distance for a 208-Byte Message

Number of Hops[a]	1	3	7	15	31	63
Size of Network[b]	4	16	64	256	1024	4096
Wormhole Routing[c]	276.2	276.4	276.8	277.6	279.2	282.4
Store-and-Forward[d]	574	1473	3272	6869	14062	28449

[a]Hops is used here to mean the number of inter-node links traversed.
[b]Size of network is based on size for which a square mesh would yield the above number of hops on the average for a corner (worst case) node.
[c]Computed analytically, based on [Seit88], adjusted for typed message empirical results.
[d]Computed analytically, based on [Gust88].

3.4 Overview

A brief overview of the algorithm follows. Detailed pseudocode is included in Appendix A. The algorithm may be segmented into five phases, each of which is disjoint for pipelining purposes:

1. Retrieval of Input Relations. The disk nodes retrieve from disk storage the local fragments of the two input relations, $R[i]$ and $S[i]$, applies selection and projection predicates, and extracts keys ($RK[i]$ and $SK[i]$), including join attributes and pointers, The local sort nodes sort the keys into join attribute order. The disk nodes then hash split the local fragments of keys into N^2 hash[6] buckets ($R[i:h]$ and $S[i:h]$) and store them in main memory or on disk.

2. Construction of Semi-join Filters. The disk nodes retrieve the sorted keys and dispatch them to the hash-value-determined merge sites. The

[6]The folded hash is based on a hash value length equal to the ceiling of the number of bytes obtained by converting $\log_2 N^2$ bits to bytes. The hash value is then obtained by adding successive lengths of the join attribute(s) until all is consumed. The hash filter is the first $\log_2 N^2$ bits of the hash value.

merge nodes join RK and SK to produce join filters. The filter fragments, $RK[ij : h]$ and $SK[ij : h]$ are then dispatched to the source cluster sort nodes, where the fragments are merged (within pipeline stages) and sorted into input distribution attribute order. The resultant sorted semi-join filters are returned to disk node i for storage.

3. Construction of Semi-joins. Disk nodes retrieve the reduced tuple sets, $R[i]$ and $S[i]$ and the semi-join filters, $RK[ij]$ and $SK[ij]$, merge the (between pipeline stage fragments) and dispatch them to the local merge nodes. The merge nodes perform a merge-join of the semi-join filter and the reduced local relation fragment to produce the semi-joins of the two relations, $R[ij]$ and $S[ij]$. The semi-joins are transmitted to the sort nodes for sorting into join attribute order and then returned to the local disk node i for storage. The local disk node will have M fragments of $R[ij]$, N fragments of $S[ij]$ or both, depending upon whether the domains of R and S overlap.

4. Performance of Partial Joins. The semi-join fragments (based on origin i and origin of join partner j) are sent to a local (within cluster of disk node i) merge node for execution of the partial join $R[ij]$ and $S[ji]$. The resultant fragment of $T[i]$, after application of selection and projection predicates is sent to the local sort node to be sorted into result distribution attribute order. The fragments of $T[i]$ are then stored on disk node i.

5. Creation of Output Relation. There are three alternatives in this phase:

   - If T is a temporary relation, the local fragments are retrieved, merged and stored on node i.
   - If T is a permanent relation, the local fragments are retrieved and split according to distribution attribute and destination. After transmission, the fragments received are merged and stored.
   - If T is to be sent to the host, the local fragments are retrieved and merged and then merged up the logical merge tree[7].

[7]The merge tree is created using a low number of levels to reduce latency in initial results by defining branches per node as equal to 2^l where l is the level of the node.

4 Experimental Design

4.1 Issues

Given the architecture, several issues of interest present themselves:

- How best to configure a machine for processing databases?
- Which algorithms to use? and
- What is expected performance?

Significant to the issue of algorithm selection were the following considerations:

1. In the environment that we are investigating, the primary performance determinant would be functioning of high-speed joins;
2. Given the high-speed sort capability described in Section 2 above, the principal means of achieving joins would be through merge-based joins of previously sorted tuples; and
3. Given the selectivity characteristics expected from relations based on standard Wisconsin database relations [Bitt83], semi-join based join algorithms would be practical.

With respect to the issue of machine configuration, recall from Sections 2 and 3 that physical processing nodes may optionally have disk(s) and/or a CADM sort engine attached, and that three logical processing nodes are identified — namely sort, disk and merge nodes. We therefore need to consider the mapping of logical node functionalities to alternative combinations of processor configurations.

We identify four physical node types:

1. A required processor with disk, that is utilized for I/O processing, and optionally for sort or merge processing or both. If utilized as a sort node, it may have a SORTER. A Type 1 node must be included in any configuration.
2. An optional diskless processor, that will be utilized for merge or sort processing, or both, or simply as a passive communication node. If utilized as a sort node, it may have a SORTER.

3. An optional diskless processor, utilized as a sort node, and attached directly to a Type 1 or Type 2 node. (The other types are attached to the interconnection network.) It may have a SORTER.

4. An optional diskless processor, without a SORTER, utilized only for merge processing.

The mapping of logical node functionalities to physical processor types in various configurations is illustrated in Table 2.

Table 2: Functionality of Network Components

Configuration	Type 1	Type 2	Type 3	Type 4
0	DMS			
1	DS			M
2	DM		S	
3	D		S	M
4	D	MS		
5	D	S		M
6	D	M	S	
7	D	C	S	M

Legend:	
	D – Disk node
	M – Merge node
	S – Sort node
	C – Communications only

In the following discussion, cluster is used to identify the smallest homogeneous partition of the network. As such, a cluster may contain one or more physical nodes.

An original option space was described as follows:

1. The number of processors of a given type within a cluster were limited to 0, 1, 2 or 4 (1, 2 or 4 for Type 1).

2. Memory size options for type 1 processors were 4, 8, 16 and 32 Mb.

3. 1 to 3 disks per processor were possible.

4. Inclusion (or not) of SORTER with 16, 64 or 256 1-Kbyte CADM chips on the sort node.

5. Networks with and without wormhole routing.

6. Clusters were limited to powers of $2 \leq 12$.

Clearly, the methodology chosen must have an explicit or implicit means of reducing this large option space.

4.2 Methodology

The methodology chosen was the following:

1. Using a commercial spreadsheet, an analytic model of the problem was created. This model is discussed in more detail in Section 5. Starting with a basic uniprocessor configuration (single 4 Mb CPU with single disk), alternative enhancements were considered. For the base case, the effects of eight enhancements were considered: Adding a type 1, 2, 3 or 4 processor; Doubling or quadrupling main memory; Adding a disk; or Adding a SORTER with 16 1-Kbyte CADM chips to the sort node. The most effective of these alternatives then becomes Step 1 in the stepwise enhancement of the processing engine.

2. The relations used in this study are based on the standard Wisconsin benchmark relations [Bitt83]. A join of two 100,000 tuple relations with result relation of 100,000 tuples was selected as being of general applicability.

3. Evaluation of the finite alternatives at each successive step utilized a weighted criterion function, loosely based on component costs. Weights used were the following:

   - CPU — 1.000;
   - Main Memory — .125 per Mb;
   - Disk — 0.500;
   - Sort Engine — 0.250; and
   - Wormhole Routing — 0.250 per node connected.

The selected alternative i for the next step is the one providing the largest value $(W_i * T_i)/(W_0 * T_0)$, where W is the weighted cost, T is the elapsed time for the join and the subscript 0 refers to the base case. Intuitively, this is seen to provide a weighted measure of scalability.

4. An interesting subset of alternatives was selected for further evaluation. This subset was simulated for different network sizes. The simulation model and simulation results are described in more detail in Sections 6 and 7. A by-product of the repetitive use of the analytic model was refinement of the join algorithm as areas for potential improvement manifested themselves. The algorithm described in Section 3 incorporates refinements suggested by the analytical modelling.

5. Results of the simulation would be used in drawing conclusions concerning the scalability of parallel joins on high performance multicomputers.

5 Analytic Model

5.1 Design

The analytic model is organized into specification, mapping and pipelining sections. The specification section is further broken down into the following subsections: problem specification, hardware specification, network specification and algorithm specification. The mapping section is concerned with mapping all the system parameters and implicit work provided in the specification section into the pipelined results section. The principal function of the pipelined results section is to convert the work mapped onto pipeline components into elapsed time. The output of the model is of two forms: work performed by component and elapsed time.

The problem specification subsection is concerned primarily with defining the problem in terms of tuple sizes, key size, relation cardinalities, selection and join specificities and disposition of join results (to host, temporary or permanent relation). The hardware specification subsection is concerned both with component performance and component configuration; thus, it includes cpu cycle times, disk RPM's and latencies, transmission times, SORTER size (and/or inclusion), network type (store-and-forward versus wormhole)

and number of disks per disk node. Network specification includes number of processors of each physical type per cluster, number of clusters, memory per processor by type and type of merge tree used for output. Whereas the preceding sections are highly parameterized, the algorithm specification is algorithm-specific, meaning that implementation of an alternative algorithm would require a major rework of the model.

The mapping section may be broken down into several levels of aggregation, culminating in the passing of work to be done and work performed currently, by hardware component, to the pipelined results section. Mapping the algorithm steps against the hardware components included in the instant configuration allows derivation of atomic operation times. Mapping atomic operation times against the problem composition determines the total amount of work to be done in terms of time and numbers of atomic operations. Network specifications are then compared with work to be done in order to determine the amount of work per node.

In the pipeline results section, comparisons of work to be done by component across a pipeline stage result in the pipeline granularity (number of pipeline cycles to be performed). This allows work to be done across the network to be converted into elapsed time. Byproducts of this process include work done by component (disk, cpu, etc.) per node per cluster.

5.2 Analytic Results

Analytic results were useful in two areas:

1. Reducing the configuration option space.

2. Quickly analyzing issues as they arose.

Table 3 shows the configurations selected for simulation. Cases A, B, C and D, correspond to the configuration of a cluster. Each was simulated for 1, 2, 4, 8 and 16 clusters. An unexpected result was the absence of multi-node clusters meeting our criteria for simulation[8].

Early results indicated that configurations which included both Type 2 and Type 4 nodes were efficient. However, in these earlier results, both sort engine and disk were weighed as 1 (equal to the CPU). We later determined

[8]Case D was included so that one multi-node configuration would be reported.

Table 3: Configurations Simulated

Case	Type 1			Type 4
	MM	SORTER	Disks	MM
A	8	no	1	N/A
B	8	yes	1	N/A
C	8	yes	2	N/A
D	8	yes	2	1
Legend:	MM – Main Memory in Megabytes			
	N/A – Not Applicable			

that a sort engine weight of .5 was the point of indifference between the addition of Type 4 nodes or additional Type 1 nodes as the network expands. The selection of Cases A, B and C was based on their being important steps in our navigation of the configuration option space, with Case C being the most efficient.

Comparisons to previously published results, illustrated in Table 4, led us to believe that we were following a productive development path, in spite of the difficulties inherent in comparing performance between systems with different configurations and levels of recovery protection. The metric Response Time is the elapsed time to deliver the last tuple to the host, while Latency refers to the elapsed time to deliver the first tuple to the host.

In analyzing scalability, it became apparent that many interacting factors affect performance, including the following:

- Disk accesses decreasing as system-wide main memory increases;
- Merges of sort buckets decrease as the size of partitions decreases (as network size increases);
- The effect of $NlogN$ processes being applied to smaller partitions as network size increases;

One way of balancing these effects is to increase the size of the relation as network size increases. The results of this "Constant Tuples per Node"

Table 4: Results Comparison: $T = R \bowtie S$, $|R| = 1,000,000$, $|S|$,$|T| = 100,000$.

Machine	Processor Type			Latency (sec)	Response Time (sec)
	With Disks	Diskless	Control		
Case C	16 – 8 Mb	none	1 - 8 Mb[a]	66	80
Case D	8 – 8 Mb	8 – 1 Mb	1 - 8 Mb	124	134
Terradata[b]	20 – 4 Mb	none	4 – 4 Mb	N/A	3144
Gamma[c]	8 – 2 Mb	8 – 2 Mb	1 – 2 Mb	N/A	2938

[a] This refers to the node mapper. An alternative configuration includes it as part of the host software.
[b] [DeWi87]
[c] [DeWi88]

analysis, when applied to Case C, are shown in Table 5. The metric "Relative Performance" is the ratio computed as the elapsed time for the base case (2 clusters and cardinality of 100,000) divided by the elapsed time for each instance. The third column (Response Time) shows results which appear to reflect poor scalability. However, when we look at the Latency metric, we see good scalability characteristics. As would be expected, when we look at the final merge[9] only, we see poor scalability. Indeed, the communications work in the final merge is growing almost as fast as the number of tuples.

Another interesting result of this analysis is contained in the last column. The Latency Pipelining Effect metric compares execution time without pipelining to execution time with pipelining. It is computed by taking the difference in the two elapsed times and dividing by time without pipelining. The algorithm described in Section 3 provides for finer pipeline granularity as disk nodes increase. Thus, the expected result is to see savings due to pipelining increase as the number of disk nodes increases. However, the granularity may become so fine that the overhead costs of partially filled communications buffers, sort devices and disk tracks will offset the benefits of having additional nodes. This column only applies to latency; the final merge is excluded from this particular analysis. Within

[9] Final merge refers to Phase 5 in the Section 3 description and approximates the difference in the response time and latency metrics.

the final merge, pipelining effects (as data moves up the merge tree) are $O(no\_of\_merge\_levels/(no\_of\_merge\_levels+1))$.

Table 5: Constant Data per Node Results: $T = R \bowtie S$, $|R| = |S| = |T|$.

Number of Clusters	Cardinality of Relations	Relative Performance				Latency Pipelining Effect
		Response Time	Latency	Final Merge	Final Merge Communication	
2	100,000	1.000	1.000	1.000	1.000	13.5 %
4	200,000	.951	.994	.821	.556	13.7 %
8	400,000	.889	1.018	.605	.294	14.1 %
16	800,000	.761	1.011	.397	.151	15.7 %
32	1,600,000	.598	1.025	.235	.076	19.1 %
64	3,200,000	.414	1.017	.130	.038	24.7 %

The use of the high-performance communications (wormhole routing) has two expected effects:

1. Improved general performance over store and forward technology.

2. Relative performance improvements for heterogeneous architectures.

Performance comparisons on the Latency metric for Case C and Case D, as shown in Table 6, support these findings, with Case C exhibiting performance gains in excess of 30 % for most network configurations when the "wormhole routing" facility is installed. Case D performance showed greater than 50 % gains by adding "wormhole routing." It is expected that the communications cost gain would be greater for case D, since both intra-cluster and inter-cluster communications occur. For both cases, results appear to be scalable upward. Response Time results were even more dramatic because of the significant effect of communications time on performance during the final phase. This was expected as noted in Section 3 and illustrated in Table 1.

6 Simulation Design and Implementation

Simulations were written in written in PAWS [Scie87]. PAWS is a specialized language for simulation of computer systems. It models systems in terms of

Table 6: Effects of Wormhole Routing: $T = R \bowtie S$, $|R| = |S| = |T| = 100,000$.

Percentage Reduction in Latency							
Number of Clusters	1	2	4	8	16	32	64
Case C	0.0	30.0	33.5	48.8	51.1	63.1	68.2
Case D	55.3	53.1	58.5	62.0	64.9	66.8	69.6

transactions and servers. Servers may be active (disks or CPU), or passive (memory). Transactions flow between servers, where they request or release resources. In our study, transactions represent units of processing such as disk tracks, sort engine bins (the number of tuples that can be sorted at one time), and inter-node messages.

As an example of simulation design, consider message passing. This aspect was of particular concern, as timing experiments on the Symult had indicated considerable operating system overhead for message passing. Transmission of a message required less than 100 microseconds and was independent of message length; it was not explicitly modeled. The time required to *receive* a L-byte message is given by the following equation:

$$T_{recv} = T_{setup} + mT_{packet} + LT_{byte}$$

where $m = \lceil L/256 \rceil$. Timing experiments on the Symult indicated that, for typed messages, T_{setup} was 250 microseconds, T_{packet} was 22 μs, and T_{byte} was 0.067 μs.

This delay is split between the sending and receiving nodes. In the simulation, a transaction that represents a message passes through two servers. First, it visits an AMRD server at the sending node; the service time is LT_{byte}. This step represents the actual transmission of data through the AMRD. The second step is a visit to the CPU at the receiving node; this represents the operating system overhead of mapping the message into the address space of the application process. The CPU service time is $T_{setup} + mT_{packet}$.

Simulation parameters were derived in several ways. Communication costs were based on experimental results. CPU costs for each operation,

such as a join, were based on consideration of the number of basic steps (comparisons, pointer copies, etc) and the speed of these steps on the MC68020 processor. As with the analytic model, it was assumed that data were uniformly distributed initially, and that all subunits created during processing (buckets, partial joins, and so on) are also of uniform size.

The disk model assumed a 32 Kbyte track size, a 28 millisecond seek time, and a transfer time of 16.7 ms; it also was assumed that a seek occurred before each track was accessed. Memory was modeled as 256 byte pages to reflect the Symult memory architecture. Size requirements were rounded up to the next page, in order to model internal fragmentation.

7 Simulation Results

The simulations assumed a join of two 100K tuple relations, each tuple being 208 bytes; the output cardinality was also 100K. The four cluster configurations specified in Table 2 were simulated for 1, 2, 4, 8, and 16 clusters. Table 7 lists response time, i.e., the elapsed time between the start of the join and delivery of the *last* result tuple to the host. Table 8 lists latency, i.e., the elapsed time until the *first* tuple arrives at the host. More detailed results are reported in Appendix B. Separate simulations were written for the first four disjoint phases of the algorithm (as described in Section 3). Analytical results are reported for Phase 5, merge of results and delivery to the host. Since the goal of this paper is to investigate scalability and the effect of cluster configuration on performance, the latency metric is used rather than response time. Phase 5 elapsed time is an inconsequential component of latency. For the configurations reported, Phase 5 latency is $< .5$ seconds, as shown in Appendix B.

One potential objection to the choice of latency as a metric is that the delivery of results to the host in phase 5, as described in Section 3 will constrain the possible speedup. However, this constraint exists only if the result relation is to be delivered to the host as a stream of merged tuples *and* the host can consume data faster than the final merge node can produce it. Even in this case, an analysis of latency is still valuable, as latency will reveal performance differences between configurations, while response time results may be dominated by host acceptance bandwidth.

In other situations, latency is closely related to response time. If phase

Table 7: Response Time Results: $T = R \bowtie S$, $|R| = |S| = |T| = 100,000$.

Number of Clusters	Elapsed Time (seconds)			
	Case A	Case B	Case C	Case D
1	953.6	553.4	311.0	310.7
2	500.8	314.3	202.5	194.1
4	267.0	155.2	106.7	92.5
8	122.8	50.5	43.3	33.6
16	59.3	29.0	25.5	21.4

Table 8: Latency Results: $T = R \bowtie S$, $|R| = |S| = |T| = 100,000$.

Number of Clusters	Elapsed Time (seconds)			
	Case A	Case B	Case C	Case D
1	896.7	496.5	282.6	282.3
2	454.6	268.0	156.3	156.8
4	239.4	127.6	79.0	73.9
8	107.2	34.8	27.6	24.4
16	47.0	16.6	13.2	11.9

5 produces a (temporary or permanent) new relation, then no single node is a bottleneck and phase 5 should scale as well as the rest of the algorithm. Another possibility is that phase 5 delivers an aggregate, such as a sum, to the host. In this case, the time to compute that aggregate will be small compared to the latency, so latency determines response time.

7.1 Scalability

Scalability is used here to mean the relationship between performance and network size and can be examined by analyzing Table 8.

Before proceeding, it is useful to define the notion of *linear* speedup. Assume a system with N clusters and an elapsed time t. If a larger system with kN ($k > 1$) clusters has an elapsed time of t/k, then it has achieved linear speedup. If the new elapsed time is less than t/k, then a *superlinear* speedup has occurred. An elapsed time greater than t/k indicates a *sublinear* speedup.

For all four configurations, the following pattern may be seen: the speedup between 1 and 2 clusters, and between 2 and 4 clusters, is sublinear. Then the speedup between 4 and 8 clusters is substantially superlinear. Between 8 and 16 clusters, the speedup is slightly superlinear.

The use of latency removes host acceptance bandwidth from the measurement of join performance. The assumption of equally distributed data precludes the potential problem of uneven loads between clusters.

While the above characteristics make it reasonable to expect linear speedup, they do not explain the superlinear speedups seen when N is increased to 8, which is caused by synergistic effects. As N increases, the total memory in the system increases, and the number of tuples processed by each cluster decreases. For $N < 8$, inputs and outputs for phases 3 and 4 must be stored on disk. For $N = 8$ and $N = 16$, all disk accesses except the initial read in phase 1 can be eliminated, substantially reducing response time.

A similar, but less substantial, effect occurs when $N = 16$. Most of the sorts performed during the join are multiple-bin sorts: data must be tagged and split among several bins, which are sorted separately and then merged. In the $N = 16$ case, many sorts require only one bin, eliminating the tagging and merging stages.

Finally, several parts of the algorithm have time requirements that are $O(m \log_2 m)$, where m is the number of tuples. When N is doubled, m de-

creases to $m/2$ and these parts of the algorithm exhibit superlinear speedup. The most prominent example of this effect is the heap sort used in Case A (in place of the sort engine).

7.2 Comparison of Cluster Configurations

The benefits of adding hardware to a cluster can be evaluated by comparing the columns of a single row in the Table 8 (since each row represents a fixed number of clusters). There are three incremental improvements reported: adding a sort engine to each cluster, adding a second disk, and adding a merge node (CPU and memory).

Comparison of Cases A and B show the benefits of the sort engine. Even in systems with 4 or fewer clusters, where the disk is often a bottleneck, the sort engine reduces response time by about 40 percent. In the 8 and 16 node systems, where tuples are buffered in memory and the sorts consume much of the CPU time, the improvement is even more pronounced – close to 70 percent.

The benefits of adding a second disk to each cluster can be evaluated by comparing response times for Cases B and C. For 1, 2 and 4 cluster systems, which must save the results of each phase on disk, adding a second disk reduces response time by about 40 percent. For the 8 and 16 node configurations the 20 percent improvement reflects a faster phase 1 (initial read and filter).

Comparing Cases C and D reveals that adding a merge node to each cluster has little effect on response time. For the 1 and 2 cluster systems, there is no improvement in response time. In this situation, processing is disk-bound rather than CPU-bound, so making more CPU cycles available does not improve system performance.

For 4 or more clusters, adding the merge node improves response time by 6 to 12 percent. As these cases are reading tuples from memory rather than disk, they are CPU-bound, so one might expect a larger improvement in response time. The small amount of improvement is explained by the high overhead of data communication: the communication cost is the same order of magnitude as the processing cost. Thus, while the additional processor off-loads the disk node CPU, much of CPU time thus saved is taken up by message passing to and from the merge node.

8 Summary and Conclusions

The research presented here leads to the following conclusions:

- Message passing multi-computers are a good environment for parallel joins, provided the communication overhead is not excessive.
- The proposed sort engine is a cost-effective way to substantially improve performance for sort-merge join processing.
- The algorithm presented and studied here is a novel approach to parallel joins that combines the advantages of hashing and semijoins.
- Analytic methods can be used both to select promising configurations for further study, and to help refine the algorithm.
- Simulation results confirm the analytic predictions.
- The proposed architecture scales very well. Delivery of merged tuples to the host is a bottleneck only if the host has a greater bandwidth than the final merge node.
- Increasing the number of clusters has a synergistic effect that can generate superlinear speedups.

Acknowledgments

The authors wish to thank Winn Biesele and Robert Van de Geijn for performing the experiments that provided data on Symult message passing performance.

References

[Bhat87] Bhat, Vivekanand, "Design of the CADM Based Sort/Search Engine", Department of Computer Sciences, Technical Report No. TR-87-36, University of Texas at Austin, September, 1987.

[Bitt83] Bitton, D., D.J. DeWitt, and C. Turbyfill, "Benchmarking Database Systems – A Systematic Approach", Proceedings of the 1983 Very Large Data Base Conference.

[Dall86] Dally, William J., "On The Performance of k-ary n-cube Interconnection Networks", Caltech Computer Science Technical Report 5228:TR:86, 1986.

[Desa88] Desai, Ameesh Yogendra, "Node Mapper, an Indexing Structure for the KYKLOS Database Machine", M.S. Engineering Thesis, Department of Electrical and Computer Engineering, University of Texas at Austin, August, 1988.

[DeWi87] DeWitt, David J., Marc Smith, Haran Boral, "A Single User Performance Evaluation of the Teradata Database Machine", MCC Technical Report Number DB-081-87, March, 1987.

[DeWi88] DeWitt, David J., Sharam Ghandeharizadeh, Donovan Schneider, "A Performance Analysis of the Gamma Database Machine", Proceedings of the 1988 SIGMOD International Conference on Management of Data, June, 1988.

[Flai87] Flaig, Charles M., "VLSI Mesh Routing Systems", Caltech Computer Science Technical Report 5241:TR:87, May, 1987.

[Gust88] Gustafson, J.L., G.R. Montry, and R.E. Benner, "Development of Parallel Methods for a 1024-Processor Hypercube", SIAM Journal on Scientific and Statistical Computing, Vol.9, No.4.

[Mene85] Menezes, B.L., and R.M. Jenevein, "KYKLOS: A Linear growth Fault-tolerant Interconnection Network," Proceedings of the International Conference on Parallel Processing, August, 1985.

[Scie87] Scientific and Engineering Software, "PAWS 3.0 Performance Analyst's Workbench System User's Manual", Austin, Texas, 1987.

[Seit88] Seitz, Charles L., Jakov Seizovic, Wen-King Su, "The C Programmer's Abbreviated Guide to Multicomputer Programming", Department of Computer Science, California Institute of Technology, Caltech–CS–TR–88–1, January, 1988.

[Symu89] Symult Systems Corporation, "Programmer's Guide to the Series 2010^{TM} System", Monrovia, California, January, 1989.

Appendix A. Pseudocode for Join Algorithm

In this distributed environment the following notation is used:

Let X be one of R, the larger input relation; S, the smaller input relation; or T, the result relation;

Then,

$X[i]$ =the fragment of X stored at store node i;

X_r =the remainder of X after selection (and/or projection) predicates are applied;

$X[ij]$ =the fragment of X stored at store node i, joinable with the fragment of Y (not X) stored at store node j. Thus, $R[ij]$ would be the tuples of R stored at i joinable with $S[ji]$

$X[:b]$, $X[ij:b]$ =the fragment hashed to bucket b.

$h = number\_of\_hash\_buckets = N^2$

$s = local\_number\_of\_sort\_buckets = local\_fragment\_cardinality/(cadm\_size * K_s)$

XK =the projection of keys and pointers of relation X

"*" is used to indicate the range of a variable, for example, the set of tuples from store node i, assigned to joins with all other nodes is represented as $X[i*]$. $X[i*] = \Sigma_{j=1,N} X[ij]$

The following pseudocode verbs are used in addition to commonly used terms:

"parbegin Y T ... parend" indicates a sequence of operations performed in parallel at Y nodes of type T (disk, sort or merge).

"pipbegin Z ... pipend" indicates a pipelined sequence of operations. These operations are subdivided into stages (see below) each of which is executed Z times and which are executable as follows: stage 1 is executed by itself in cycle 1; in cycle 2, stage 1 and stage 2 are executed; then cycle 3, composed of stages 1, 2 and 3; etc.

"stage A" indicates the beginning of a pipeline stage. Statements following are included until the current stage is ended by another stage statement or a pipend. A is the stage number and is included for reading ease as oppose to semantic necessity. ... stagend is used only when necessary for unambiguous pseudocode.

comments are enclosed in quotes ("comments").

```
begin "join"

    "Phase 1–Retrieval of Input Relations"

    "Pipelines 1 and 2 correspond to subphase 1A; pipelines 3 and 4 correspond
    to suphase 1B. If the domain of disk nodes for R and S do not overlap,
    subphases 1A and 1B can be executed in parallel."

    pipbegin |RK[i]'|/cadm_size "1"
        stage 1
            parbegin N disk
                retrieve local fragment of R, R[i], from disk
                apply selection and projection predicates creating R[i]'
                extract join attributes from R[i]' creating RK[i]'
                if R[i]' ≠ R[i] and memory full, store on disk
                transmit RK[i]' to local (within cluster) sort node
            parend
            parbegin K_s * N sort
                for lcv = 1 to 1/K_s begin
                    receive RK[i]'
                    tag sort buckets of RK[i]'
                forend
            parend
        stage 2
            parbegin K_s * N sort
                for lcv = 1 to 1/K_s begin
                    sort buckets for disk node i
                forend
            parend
    pipend
    pipbegin |RK[i]'|/comm_buffer_size "2"
        stage 1
            parbegin K_s * N sort
                for lcv = 1 to 1/K_s begin
                    merge sort buckets
                    transmit merged keys to local disk node
```

```
            forend
        parend
    stage 2
        parbegin N disk
            receive sorted RK[i]'
            using folded hash algorithm, hash RK[i]' to N^2 buckets,
                    RK[i : b]', b = 0 to N^2 - 1
            store hash fragments
        parend
pipend
pipbegin |SK[i]'|/cadm_size "3"

    (pseudocode identical with pipeline "1", except for substitution
            of S for R and M for N, where appropriate)

pipend
pipbegin |SK[i]'|/comm_buffer_size "4"

    (pseudocode identical with pipeline "2", except for substitution
            of S for R and M for N, where appropriate)

pipend

"Phase 2–Construction of Semi-join Filters"

pipbegin N/K_m "5"
    stage 1
        parbegin N disk
            for lcv = 1 to K_m * N begin
                transmit fragments RK[i : h]' to merge nodes h
            forend
        parend
        parbegin M disk
            for lcv = 1 to K_m * N begin
                transmit fragments SK[i : h]' to merge nodes h
            forend
```

```
        parend
        parbegin K_m * N merge
            for lcv = 1 to N
                receive RK[i]' from disk node i
            forend
            for lcv = 1 to M
                receive a fragment SK[ij]' from disk node i
            forend
        parend
stage 2
        parbegin K_m * N merge
            perform merge join on M + N fragments RK[i : b]' and SK[j : b]',
                    resulting in M + N fragments of joinable keys: RK[i* : b]'
                            and SK[j* : b]' at each merge node
                    for lcv = 1 to N
                        transmit joinable key fragment RK[ij : b]'
                                to cluster of origin sort node
                    forend
                    for lcv = 1 to M
                        transmit joinable key fragment SK[ij : b]'
                                to cluster of origin sort node
                    forend
                parend
                parbegin K_s * N sort
                    for lcv = 1 to K_m * N/K_s
                        receive fragments of RK[i* : b]' from merge nodes
                        if needed then tag RK[i* : s : b]'
                    forend
                parend
                parbegin K_s * M sort
                    for lcv = 1 to K_m * N/K_s
                        receive fragments of SK[i* : h]' from merge nodes
                        if needed then tag SK[i* : bh]'
                    forend
                parend
            stage 3
```

parbegin $K_s * N$ sort
sort s buckets of $RK[i* : b]'$
merge buckets and transmit to node i
parend
parbegin $K_s * M$ sort
sort s buckets of $SK[i* : ccv]'$
merge buckets and transmit to node i parend
parbegin N disk
receive $RK[i* : ccv]'$
parend
parbegin M disk
receive $SK[i* : ccv]'$
parend
stage 4
parbegin N disk
store $RK[i* : ccv]'$
parend
parbegin M disk
store $SK[i* : ccv]'$
parend
pipend

"Phase 3. Construction of Semi-joins"

"Pipeline 6 corresponds to subphase 3A; pipeline 7 corresponds to suphase 3B. If the domain of disk nodes for R and S do not overlap, subphases 3A and 3B can be executed in parallel."

pipbegin $|RK[i*]'|/cadm\_size$ "6"
stage 1
parbegin N disk
retrieve and merge fragments of $RK[i*]'$
transmit $RK[i*]'$ to local merge node
retrieve $R[i]'$
transmit $R[i]'$ to local merge node
parend

parbegin $K_m * N$ merge
for lcv = 1 to $1/K_m$
receive $RK[i*]'$ from local disk node
receive $R[i]'$ from local disk node
forend
parend
stage 2
parbegin $K_m * N$ merge
for lcv = 1 to $1/K_m$ begin
perform a merge join of $RK[i*]'$ and $R[i]'$, resulting
in sort bucket b of semi-join tuples, $R[i* : b]'$
forend
parend
parbegin $K_s * N$ sort
for lcv = 1 to $1/K_s$ begin
receive sort buckets, $R[i* : b]'$
forend
parend
stage 3
parbegin $K_s * N$ sort
for lcv = 1 to $1/K_s$ begin
sort buckets, $R[i* : b]'$ into join attribute order
transmit semi-join tuples to node i
forend
parend
parbegin N disk
receive sorted fragments $R[i* : b]'$
parend
stage 4
parbegin N disk
split sorted fragments into M partitions, $R[ij : b]'$
store the sorted fragments by join partner node,
$R[ij : b]'$
parend
pipend
pipbegin N/K_s "7"

(pseudocode identical with pipeline "6", except for substitution of S for R and M for N, where appropriate)

pipend

"Phase 4. Performance of Partial Joins"

pipbegin M/K_m "8"
stage 1
parbegin N disk
retrieve and merge the sorted fragments $R[ij:*]'$ into $R[ij]'$
transmit (on request) $R[ij]'$ to local merge node
parend
parbegin M disk
for lcv = 1 to N/M begin
retrieve and merge the sorted fragments of $S[ij:*]'$ into $S[ij]'$
transmit, as requested $S[ij]'$
forend
parend
parbegin $K_m * N$ merge
for lcv = 1 to $1/K_m$
receive $R[ij]'$
receive $S[ji]'$
forend
parend
stage 2
parbegin $K_m * N$ merge
for lcv = 1 to $1/K_m$
merge-join $R[ij]'$ and $S[ji]'$ to form the result relation fragment, $T[ij]$
apply selection and projection predicates to $T[ij]$, forming $T[ij]'$
transmit $T[ij]'$ to the local sort node

```
            forend
        parend
        parbegin K_s * N sort
            for lcv = 1 to 1/K_s
                receive T[ij]'
            forend
        parend
    stage 3
        parbegin K_s * N sort
            for lcv = 1 to 1/K_s
                sort T[ij]' into result distribution order
                transmit T[ij]' to local disk node
            forend
        parend
        parbegin N disk
            receive T[ij]'
        parend
    stage 4
        parbegin N disk
            store T[ij]'
        parend
pipend

"Phase 5. Creation of Output Relation"

if result relation T is temporary begin
    parbegin N disk
        retrieve all fragments T[i*]'
        merge into T[i]
        store T[i]
    parend
elseif result relation is to be permanent begin
    pipbegin |T[i]|/buffer_size "9b"
        stage 1
            parbegin N disk
                retrieve all fragments T[i*]'
```

split $T[i*]'$ by distribution node into $T[i* : d]'$
merge into $T[i : d]$
transmit $T[i : d]$ to distribution node
receive $T[* : d]$ at distribution node
parend
parbegin N disk
merge fragments $T[* : d]$ into $T[i]$
store $T[i]$
parend
pipend
elseif result relation is to be sent to the host begin
pipbegin $|T[i]|/buffer\_size$ "9c"
stage 1
parbegin N disk
retrieve all fragments $T[i*]'$
merge into $T[i]$
transmit to logical parent branch node
$k_1 = 1$
$k_2 = 1$
$k_5 = 1$
parend
stagend
while $k_1 * k_2 < N$
stage k_5
parbegin k_1 merge
$k_4 = k_2 * 2$
$k_3 = k_1 * k_4$
$k_5 = k_5 + 1$
if $k_1 = 1$
receive tuples from 2 nodes
merge tuple streams
send to host
else
receive tuples from $lsr(N, k_3)$ nodes
merge tuple streams
send to parent

ifend
$k_1 = k_3$
$k_2 = k_4$
parend
stagend
whilend
pipend
ifend
end "join"

Appendix B. Detailed Simulation Results

This appendix provides more detailed results for the simulations described in Sections 6 and 7. Separate results are presented for each of the first four phases of the join algorithm, as described in section 3. The first subsection of this appendix presents detailed latency and response time results. The other subsections report utilization for the four servers: disk node CPU, disk, sort engine and merge node CPU.

N refers to the number of clusters. For Cases A, B and C, this is the number of nodes, as there is one disk/sort/merge node per cluster; for Case D, there is one disk/sort node and one merge node per cluster. Times reported are in seconds.

B–1. Response and Latency Results

Tables 9 to 12 present the time required to complete each phase of the join algorithm. The times for phases 1–4 are simulation results, while the time for phase 5 was obtained analytically. Latency, or is the elapsed time for the first tuple to reach the host, is approximated by the sum of the elapsed times for phases 1–4. Response Time, or elapsed time for the last tuple to reach the host, is the sum of the elapsed times for phases 1–5.

Table 9: Latency and Response Time — Case A

N	Phase 1	Phase 2	Phase 3	Phase 4	Total 1 – 4	Phase 5		Total	
						latency	response	latency	response
1	314.08	202.55	193.66	186.31	896.60	0.09	56.97	896.69	953.57
2	158.06	87.69	100.64	107.93	454.32	0.21	48.30	454.53	502.62
4	77.29	76.16	54.06	31.64	239.15	0.22	29.52	239.37	268.67
8	36.65	32.80	19.77	17.78	107.00	0.15	17.36	107.15	124.36
16	17.02	13.79	9.88	6.06	46.75	0.16	14.03	46.91	60.78

Table 10: Latency and Response Time — Case B

N	Phase 1	Phase 2	Phase 3	Phase 4	Total 1 – 4	Phase 5		Total	
						latency	response	latency	response
1	145.18	32.87	193.10	125.28	496.43	0.09	56.97	496.52	553.40
2	78.77	18.61	100.13	70.30	267.81	0.21	48.30	268.02	316.11
4	40.19	7.10	53.64	26.39	127.32	0.22	29.52	127.54	156.84
8	19.63	4.70	4.74	5.57	34.64	0.15	17.36	34.79	52.00
16	9.51	2.10	2.40	2.43	16.44	0.16	14.03	16.60	30.47

Table 11: Latency and Response Time — Case C

N	Phase 1	Phase 2	Phase 3	Phase 4	Total 1 – 4	Phase 5		Total	
						latency	response	latency	response
1	84.83	32.87	96.79	68.06	282.55	0.04	28.49	282.59	311.04
2	48.96	18.61	50.30	38.18	156.05	0.21	48.30	156.26	204.35
4	25.63	7.10	26.82	19.23	78.78	0.22	29.52	79.01	108.30
8	12.48	4.70	4.74	5.57	27.49	0.15	17.36	27.64	44.85
16	6.02	2.10	2.40	2.43	12.95	0.16	14.03	13.15	26.98

Table 12: Latency and Response Time — Case D

N	Phase 1	Phase 2	Phase 3	Phase 4	Total 1 – 4	Phase 5		Total	
						latency	response	latency	response
1	84.83	35.45	96.87	65.08	282.23	0.05	28.49	282.28	310.72
2	48.96	18.61	50.73	38.34	156.64	0.20	37.82	156.84	194.46
4	25.63	6.38	27.33	14.38	73.72	0.20	19.01	73.92	92.73
8	12.48	3.15	4.60	4.01	24.24	0.13	10.56	24.37	34.80
16	6.02	1.24	2.35	2.19	11.80	0.13	10.63	11.93	22.43

B–2. Disk Node CPU Utilization

Tables 13 to 16 present simulation results for CPU utilization on the disk node (type 1 node). The total utilization is the time-weighted average of the utilization of each phase. If T_i is the duration of phase i and U_i is the utilization observed during phase i, then

$$U_{total} = \frac{T_1U_1 + T_2U_2 + T_3U_3 + T_4U_4}{T_1 + T_2 + T_3 + T_4}$$

Table 13: Disk Node CPU Utilization % — Case A

N	1	2	3	4	Total
1	62.3	100.0	81.6	59.6	74.4
2	63.0	51.1	78.5	52.0	61.5
4	63.1	51.7	73.1	85.2	64.7
8	63.0	51.9	99.9	90.3	71.0
16	63.9	51.4	99.9	90.2	71.2

Table 14: Disk Node CPU Utilization % — Case B

N	1	2	3	4	Total
1	15.5	88.3	19.4	32.1	26.0
2	23.1	93.8	18.7	24.7	26.8
4	26.4	84.5	17.4	34.3	27.5
8	27.2	57.9	99.3	92.2	51.7
16	28.4	25.3	97.1	88.9	47.0

B–3. Disk Utilization

Tables 17 to 20 present disk utilization results. For cases C and D, where there are two disks per cluster, the average utilization of the two disks is shown. Total utilization follows the definition given previously. A utilization of 0.0 indicates that the disk was not used during that phase.

Table 15: Disk Node CPU Utilization % — Case C

N	1	2	3	4	Total
1	26.5	88.3	38.6	59.1	45.7
2	37.1	93.8	37.1	45.1	45.8
4	41.4	84.5	34.8	47.0	44.4
8	42.8	57.9	99.3	99.2	66.6
16	44.7	25.3	97.1	88.9	59.6

Table 16: Disk Node CPU Utilization % — Case D

N	1	2	3	4	Total
1	26.5	57.3	9.5	26.9	24.6
2	37.1	86.6	8.9	14.5	28.3
4	41.4	78.5	9.8	20.9	28.9
8	42.8	73.2	24.8	52.4	44.9
16	44.7	27.6	24.3	29.0	35.9

Table 17: Disk Utilization % — Case A

N	1	2	3	4	Total
1	38.9	0.0	99.6	66.3	48.9
2	38.6	0.0	99.4	60.2	49.7
4	39.6	0.0	99.2	80.3	45.8
8	41.7	0.0	0.0	0.0	14.3
16	45.0	0.0	0.0	0.0	16.4

Table 18: Disk Utilization % — Case B

N	1	2	3	4	Total
1	84.1	0.0	99.9	98.6	88.3
2	77.5	0.0	99.9	92.3	84.4
4	76.1	0.0	99.9	96.2	86.0
8	77.8	0.0	0.0	0.0	44.1
16	80.5	0.0	0.0	0.0	46.6

Table 19: Disk Utilization % — Case C

N	1	2	3	4	Total
1	72.0	0.0	99.6	90.7	77.6
2	62.4	0.0	99.4	85.0	72.4
4	59.6	0.0	99.9	94.5	76.5
8	61.1	0.0	0.0	0.0	27.7
16	63.3	0.0	0.0	0.0	29.4

Table 20: Disk Utilization % — Case D

N	1	2	3	4	Total
1	72.0	0.0	99.5	94.9	77.7
2	62.4	0.0	99.2	84.7	72.4
4	59.6	0.0	98.6	88.3	74.5
8	61.1	0.0	0.0	0.0	31.5
16	63.3	0.0	0.0	0.0	32.3

B–4. Sort Engine Utilization

Tables 21 to 23 present results for sort engine utilization. Case A is excluded since the sort engine is not part of that configuation. Note that the specialized sort components are used in two ways: to sort bins (offline sort), and in concert with the disk node CPU to perform merges of sorted bins. These tables show the utilization of the sort engine for both tasks.

Table 21: Sort Device Utilization % — Case B

N	1	2	3	4	Total
1	5.2	67.6	5.3	8.9	10.3
2	4.9	89.3	5.2	3.1	10.4
4	4.9	78.9	4.8	5.2	9.0
8	5.4	62.8	62.8	39.3	26.5
16	6.0	51.5	62.0	46.1	25.9

Table 22: Sort Device Utilization % — Case C

N	1	2	3	4	Total
1	8.9	67.6	19.1	17.2	21.2
2	7.9	89.3	19.4	5.7	20.8
4	7.7	78.9	22.1	8.0	19.1
8	8.5	62.8	62.8	39.3	33.4
16	9.4	51.5	62.0	46.1	32.9

B–5. Merge Node CPU Utilization

Table 24 shows utilization results for the merge node CPU, which is present only in case D. Note that the merge node is not used in phase 1.

Table 23: Sort Device Utilization % — Case D

N	1	2	3	4	Total
1	8.9	62.7	10.6	17.0	18.1
2	7.9	89.3	8.9	5.6	17.3
4	7.7	87.2	8.2	9.1	15.0
8	8.5	93.1	27.8	35.7	27.7
16	9.4	81.0	27.2	26.7	23.7

Table 24: Merge Node CPU Utilization % — Case D

N	1	2	3	4	Total
1	0.0	37.2	37.0	46.4	28.1
2	0.0	10.6	33.9	39.5	21.9
4	0.0	20.5	31.5	52.9	23.8
8	0.0	15.2	97.3	94.4	36.1
16	0.0	13.5	95.1	86.3	36.4