

Massively Parallel Relational Database Processing on the Connection Machine CM-2

Masaru Kitsuregawa, Kazuhiko Matsumoto
kitsure@tkl.iis.u-tokyo.ac.jp, matsu@tkl.iis.u-tokyo.ac.jp

Institute of Industrial Science, University of Tokyo

Abstract

Several research effort has been devoted to increase the performance of the relational database systems. Various types of database machines were so far proposed. Among them, shared nothing parallel database system such as Teradata, GAMMA, TANDEM are actually implemented and detail performance evaluation is undergoing. On the other hand, recently massively parallel processor like the Connection Machine, attracts strong attention for the future super parallel system. Such parallel system is called "Data Parallel Computer" exploiting not procedure level parallelism but data level parallelism. One of the most promising application for such Data Parallel Machine be a relational database processing. However so far there has been no research to examine the effectiveness of massively parallel machine for RDB processing. In this paper we propose the two relation join algorithms for data parallel machine, that is, Data Parallel Sort Merge Join and Data Parallel Hash Join. These two algorithms are implemented on the Connection Machine CM-2. Its performance is reported.

1 Introduction

So far several kinds of research efforts has been devoted to accelerate the performance of relational database system [1, 2]. One approach is to use the high speed hardware sort engine, since sorting is one of the most fundamental operation and heavy relational operation can be performed in linear time once the relation is sorted over the key field. The sorter by Univ. of Tokyo [3], RDBM [4], IDP [5] take this approach.

The other approach is to utilize hash algorithm. Since 1983, several kinds of hash based join algorithms such as GRACE hash [6], Hybrid Hash [7],

have been proposed. These algorithms suits for parallel processing, since the buckets generated by the hash function is independent each other, and can be processed by multiple processors in parallel without any inter processor communication. Several types of parallel processor such as bus connected multiprocessor, hypercube parallel processor, tree-connected machines, are used to implement hash based parallel database processing. Functional Disk System [8, 9], GAMMA [10, 11], Teradata[12], belongs to this category. Through these efforts, relational operation has been accelerated very much so far.

From the computer architecture point of view, recently several kinds of parallel processing system is being developed. Largely, the parallel system can be classified into two categories, MIMD multi-computer which include, hypercube (iPCS, NCUBE), multistage interconnected machine (BBN butterfly), bus coupled shared memory multiprocessor (Sequent, Aliant), and SIMD array machines which include the Connection Machine, and MPP.

As described before, currently MIMD type parallel machine are extensively being researched using hash based parallel algorithm e.g. GAMMA, FDS, Teradata. On the other hand, no research has done so far for massively parallel SIMD machine such as the Connection Machine, as far as the authors know.

Currently, the Connection Machine, CM-2 embeds, 2K Weiteck floating point accelerators and is now mainly being utilized for scientific number crunching application. In this paper, we try to clarify the effectiveness of the massively parallel SIMD machine for the relational database processing [13].

2 Brief Introduction of the Connection Machine

2.1 Architecture of the Connection Machine

The Connection Machine [14] employees highly parallel architecture called massively data parallel. It has 65,536 PEs, each consisting of bit-serial processor and local memory, and high speed communication network with hypercube topology.

The Connection Machine is controlled by the front end computers attached to it by bus interface. The

front-end broadcasts instructions to all PEs of the Connection Machine through the special hardware called the Sequencer. In this way, all PEs receive the same instruction stream, but each PE has the choice whether it executes instructions or not based on its local status. This style of architecture is also called SIMD (Single Instruction stream Multiple Data stream) architecture.

2.2 Programming the Connection Machine

To program the Connection Machine, the PARIS interface is used. PARIS is the abbreviation of PARAllel Instruction Set for programming the Connection Machine system. It is a low-level library through which the actions of Connection Machine processors are directed by the front-end computer. It provides a large number of operations similar to the machine-level instruction set of an ordinary computer. The PARIS interface consists of a set of functions, macros, and variables to be called from user code. Several different versions of the user interface are provided: one for the Fortran, one for C (called C/Paris), and one for Lisp.

The names of Connection Machine primitives appearing in the algorithm descriptions of this paper are the function-names of C/Paris library provided by the Thinking Machines Corporation.

2.3 Basic programming concepts

In this section, we summarize fundamental programming concepts required to understand the descriptions of algorithms in this paper.

VP

An important feature of the Connection Machine architecture is its scalability. In most cases the same software can be executed without any change on Connection Machine systems with different numbers of physical processors. Using twice as many physical processors, a problem will run in half the time.

PARIS enhances this scalability by presenting to the user an abstract machine of the CM hardware. The most important feature is the "virtual processor" (in short, called VP) facility, whereby each physical processor simulates some number of virtual processors. A program can be written assuming any number of virtual processors; these virtual processors are then mapped onto physical processors in run-time. For example, when a PARIS add instruction is executed, each physical processor may perform many addition operations, one for each virtual processor that is mapped into that physical processor.

VPset

The set of all virtual processors associated with a data set is called a virtual processor set, or VPset. Because a single problem may be composed of more than one data set, PARIS allows for the simultaneous existence of more than one VPset. For example, relational database operations usually manipulate more than one relation. In such a case, each relation is mapped onto one VPset. Under the assumption that the size of each VPset is proper, each relation can get the space that is best fit to itself, even if the sizes of relations are different.

VPset can be dynamically allocated and deallocated during run-time. The size of the VPset is specified when VPset allocation is requested.

Each VPset defines a virtual processor ratio (in short, called VPratio). The VPratio indicates how many times each physical processor must perform a certain task in order to simulate the appropriate number of virtual processors. Accordingly VPratio equals to the number of VPs the VPset has divided by the number of physical processors.

Field

At the time of its creation, a VPset has no associated memory. PARIS provides functions to allocate and deallocate memory space for a VPset.

Memory is handled in units called Field. Conceptually, a Field is simply some number of consecutive bits of memory at the same location in every processor. A Field can be of any size. When a Field is allocated, its size is specified by the programmer.

Every Field belongs to exactly one VPset. When we speak of allocating a field to a VPset, we mean allocating a field to each VP in the VPset.

3 Join Operation

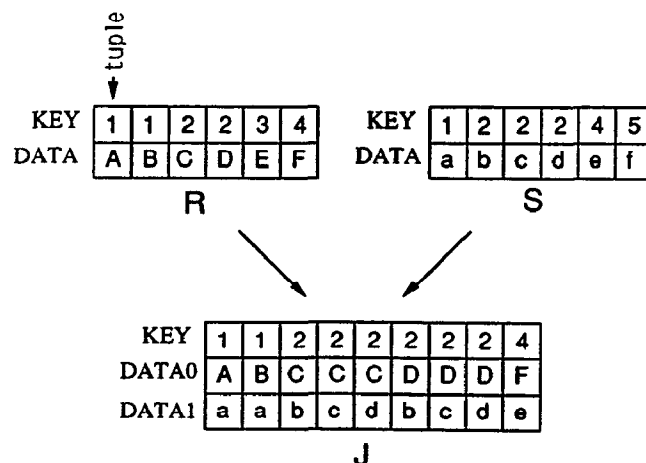


Figure1. An example of Join Operation

Join operation is one of the most important operations for relational database processing, with two relations R and S producing one relation J as shown in Figure 1. In this operation, at each key-value, corresponding tuples belonging to R and S produce cartesian products to belong to J.

If the exhaustive matching algorithm, brute force approach, is adopted, the cost of join operation is proportional to $O(N \times M)$, where N, M are the cardinalities of two relations. Thus the load of join is very heavy, and so far several kinds of algorithms has been developed to accelerate its computation speed. Two well known approaches are sort merge join and hash join. Following we modify the conventional these two algorithms to fit the data parallel machines: Data Parallel Sort Merge Join and Data Parallel Hash Join.

On the Connection Machine, each tuples are stored in its own processing element (PE), and instructions are executed by every PE in parallel. When interactions between tuples are needed, messages are exchanged by every PE in parallel. Details are described in the next sections.

4 Data Parallel Sort Merge Join on the Connection Machine

4.1 Algorithm

Data Parallel Sort Merge join processing is roughly divided into three phases as shown below.

1. Sort Merge Phase
2. Generation of Temporal Joined Relation Phase
3. Tuple Permutation Phase

We describe each of three in detail. A parallel variable, having multiple values stored over PEs, is notated by italic like *parallel-variable*.

1. Sort Merge Phase

In the first place, two source relations R and S are jointly sort-merged into one relation called M. Relation M is only temporarily generated in order to prepare for the final relation J. In this relation M, tuples are sorted in increasing order. Tuples which have the same key-value are arranged so that the tuples from R are followed by the tuples from S.

The algorithm of this phase is defined as below.

i) All tuples of R and S allocate 1-bit flag *relation-id*. The tuples of R store 0 in this flag, and the tuples of S store 1 in it.

ii) R and S are simply concatenated so that R is followed by S. Concatenation is accomplished by message sending mechanism of the Connection Machine. At this time, data portion of the tuples are replaced with the pointers representing the proces-

sor addresses for the efficiency of the following operations.

iii) Figure 2 shows the contents of the CM memory after concatenation. In this figure, each column is corresponding to a tuple (or PE), and each row represents a parallel variable (or Field). For example, the 3rd PE's value held in the Field of *sort-key* is 4. The value of sort-key is arranged according to the following formulation.

$$\text{'value'} = \text{'key attribute'} \times 2 + \text{relation-id}$$

In other words, one bit meaning relation-id are concatenated to the original key value.

	000	100	200	200	300	300	400	400	500	500	600	600	700	700	800	800
KEY	1	1	2	2	3	4	1	2	2	2	4	5				
DATA	000	001	010	011	100	101	000	001	010	011	100	101				
relation-id	0	0	0	0	0	0	1	1	1	1	1	1				
sort-key	2	2	4	4	6	8	3	5	5	5	9	11				
	M															

Figure2. An example of concatenated relation

iv) On the basis of the sort-key, tuples of both relations are sorted (Figure 3).

	1	1	1	2	2	2	2	2	3	4	4	5				
KEY	1	1	1	2	2	2	2	2	3	4	4	5				
DATA	000	001	000	010	011	001	010	011	100	101	100	101				
relation-id	0	0	1	0	0	1	1	1	0	0	1	1				
	M															

Figure3. Sorting of M relation by sort-key

We now have relation M.

2. Generation of Temporal Joined Relation Phase

Using relation M as the working place, result relation J is generated. First, we allocate the space for result relation J into which data are inserted. The process of this phase consists of data sending and data copying.

The algorithm is defined as below.

i) Each tuple belonging to M counts the number of the tuples that have the same key-value as itself. The number of tuples from R is stored in *R-tuple-number* and the number of tuples from S is stored in *S-tuple-number* (Figure 4).

ii) Each tuple of M calculates how many tuples which have the same key-value as itself will be generated in J, and stores it into *gen-tuple-number*. In other words, *R-tuple-number* multiplied *S-tuple-number* is stored into *gen-tuple-number*. Tuples whose *gen-tuple-number* is zero are no longer neces-

sary. So they are deactivated and don't participate succeeding process any more.

iii) All tuples belonging to M allocate 1-bit flag called *is-key-top*. PEs which are the top of each key-value set this flag, otherwise clear it. Aggregate sum of *gen-tuple-number* of the PEs which *is-key-top* are set is then calculated. This number is the total tuple number of J, and it determines the size of the space allocated for generating J.

KEY	1	1	1	2	2	2	2	2	3	4	4	5
DATA	000	001	000	010	011	001	010	011	100	101	100	101
relation-id	0	0	1	0	0	1	1	1	0	0	1	1
R-tuple-number	2	2	2	2	2	2	2	2	1	1	1	0
S-tuple-number	1	1	1	3	3	3	3	3	0	1	1	1
gen-tuple-number	2	2	2	6	6	6	6	6	0	1	1	0
is-key-top	1	0	0	1	0	0	0	0	1	0	0	0

M

9

Figure4. Calculation of the cardinality of result relation

iv) Two parallel variables *send-base-adrs* and *send-offset-adrs* are setup in order to calculate the address which are used to send data from M to J. PEs whose *is-key-top* are set are activated. Then they calculate exclusive accumulated sums of *gen-tuple-number* and store them to *send-base-adrs* (Figure 5).

KEY	1	1	1	2	2	2	2	2	3	4	4	5
DATA	000	001	000	010	011	001	010	011	100	101	100	101
relation-id	0	0	1	0	0	1	1	1	0	0	1	1
gen-tuple-number	2	2	2	6	6	6	6	6	0	1	1	0
is-key-top	1	0	0	1	0	0	0	0	1	0	0	0
send-base-adrs	0			2					8			

M

Figure5. Calculation of destination address (1)

Tuples whose *is-key-top* are zero setup the parallel variable *send-base-adrs* by copying the values of tuples which have the same key-value. In each group having the same key-value, *send-offset-adrs* of tuples from R and tuples from S is setup to store arithmetic series beginning from zero, having equal difference *S-tuple-number* and *R-tuple-number* respectively. All tuples of M store the sum of *send-base-adrs* and *send-offset-adrs* into *send-adrs*.

v) Tuples of M whose *relation-id* are 1 setup *major-perm-offset* and *minor-perm-offset* which are used in Tuple Permutation Phase described subse-

quently. *major-perm-offset* stores the value of *S-tuple-number*, and *minor-perm-offset* stores the serial number in each group (Figure 6).

KEY	1	1	1	2	2	2	2	2	3	4	4	5
DATA	000	001	000	010	011	001	010	011	100	101	100	101
relation-id	0	0	1	0	0	1	1	1	0	0	1	1
is-key-top	1	0	0	1	0	0	0	0		1	0	
send-base-adrs	0	0	0	2	2	2	2	2		8	8	
send-offset-adrs	0	1	0	0	3	0	2	4		0	0	
send-adrs	0	1	0	2	5	2	4	6		8	8	
major-perm-offset				1			3	3	3		1	
minor-perm-offset				0			0	1	2		0	

M

Figure6. Calculation of destination address (2)

vi) Tuples of M whose *is-key-top* are 1 send their own key attributes according to the values of *send-adrs*. Next, All tuples of M send their own tuple data according to *send-adrs*. Only tuples whose *relation-id* are 1 also send *major-perm-offset* and *minor-perm-offset* as well as data (Figure 7).

KEY	1		2							4	
DATA0	000	001	010				011			101	
DATA1	000		001		010		011			100	
major-perm-offset	1		3		3		3			1	
minor-perm-offset	0		0		1		2			0	

J

Figure7. KEY and DATA are sent to J

3. Tuple Permutation Phase

In the final phase, tuple data are permuted to proper position.

i) In relation J, KEY and DATA are copied as needed. It means that the vacant slots are filled with the copy of the valid left slot (figure 8).

At the point data sending is done, all data locating are not appropriate. In order to get desired relation, permutation of data sent from S is needed. This is done using *major-perm-offset* and *minor-perm-offset* prepared during the second phase.

ii) The top tuples of each key-value allocate parallel variable *perm-base-adrs* and store their own addresses to it. Other tuples copies the value of *perm-base-adrs* of tuples that have the same key attributes as themselves.

iii) Within each group divided by DATA0, all tuples calculate the cumulative sum of *major-perm-offset* and write them back to *major-perm-offset*.

KEY	1	2	2	2	2	2	2	4
DATA0	000	001	010	010	010	011	011	101
DATA1	000	001	001	001	010	010	011	100
<i>major-perm-offset</i>	1	3	3	3	3	3	3	1
<i>minor-perm-offset</i>	0	0	0	0	1	2	2	0

J

Figure8. Valid data are copied to vacant slots

iv) The sum of three numbers, *perm-base-adrs* and *major-perm-offset* and *minor-perm-offset*, is stored to *perm-send-adrs*.

v) According to *perm-send-adrs*, data from S are permuted.

vi) Because data are represented as pointers at this point, entities of data are retrieved from R and S (Figure 9).

KEY	1	1	2	2	2	2	2	4
DATA0	A	B	C	C	C	D	D	F
DATA1	a	a	b	c	d	b	c	e

J

Figure9. Join Operation is completed

Thus through the above three phases, final joined relation is derived as J at Figure 9, in data parallel fashion.

4.2 Implementation

In this section, we describe how the algorithm of Data Parallel Sort Merge Join described above is implemented on the Connection Machine.

Each relation R, S, M and J are represented as VPset, and each parallel variables are represented as Field. VPset and Field can be dynamically allocated and deallocated during run-time.

The operations of parallel variables such as adding or multiplying are executed locally in all PEs of the Connection Machine in parallel. On the other hand, those operations such as getting aggregate sum of all PEs' values are executed by the whole machine under the cooperation of all PEs.

In the following, we describe implementation details on each phase.

1. Sort Merge Phase

First, three VPset are allocated by the primitive **CM-allocate-heap-field** to represent relation R, S and M. Given the PE array size, the number of

tuples of R and S determines the optimal size of corresponding VPset, and the sum of them determines the optimal size of VPset of M. However, in order to measure the effect of VPratio, we can change VPset size.

After VPset are allocated, R and S are loaded to corresponding VPset. Tuples are loaded from the front end machine in serial fashion.

Relation concatenation is implemented as tuple sending operation from R and S to M. The **CM-send** primitive is used to send tuples. When this primitive is invoked, each PE sends its own message to the arbitrary destination PE simultaneously. At this point, only KEY and address are transferred, data body is not moved.

Making sort-key is implemented by those primitives such as **CM-add** or **CM-multiply**. These primitives are locally executed by all PEs in parallel.

Sorting is implemented by the primitive **CM-rank**, which reports the rank of sort-key value that each PE has. This operation is performed using communication network of the CM hardware efficiently.

2. Generation of Temporal Joined Relation

To get R and S-tuple number, the **CM-scan-with-add** primitive is used. This primitive is also used for address calculation. Since this primitive is implemented efficiently by using the CM communication network directly, this operation can be executed fast.

For calculating *gen-tuple-number*, all PEs execute multiplication at the same time by using the **CM-multiply** primitive. So 8,192 (the number of the physical processors, if CM-2 is full system, this is 65,536) multiplications are performed exactly in parallel.

The size of generating relation J is unknown at the beginning, and figured out in this phase. So the decision of the size of VPset corresponding J and its allocation is done at this point. To calculate the size of J, the primitive **CM-global-add** is used. This is also implemented by using the CM communication network. It accumulates the value stored in *gen-tuple-number* of all active PEs.

Sending data from M to J is implemented by the primitive **CM-send**.

3. Data Permutation Phase

Copying data in J is implemented by the primitive **CM-scan-with-copy**.

Again, the primitives **CM-send** and **CM-scan-with-add** are used to implement data permutation operation.

In the process of calculating destination address, all PEs perform additions in parallel.

5 Data Parallel Hash Join on the Connection Machine [13]

5.1 Algorithm

Data Parallel Hash join processing is roughly divided into three phases as shown below.

1. Parallel Dynamic Clustering Phase
2. Local Nested Loop Join Phase
3. Joined Relation Generation Phase

We describe in detail about each of three in following. A parallel variable is notated by italic like *parallel-variable*.

1. Parallel Dynamic Clustering Phase

In the first place, tuples are partitioned into multiple clusters by hashing in Data Parallel Hash Join algorithm. Clustering is done so that tuples in the same cluster have the same hashed value. The algorithm is defined as below.

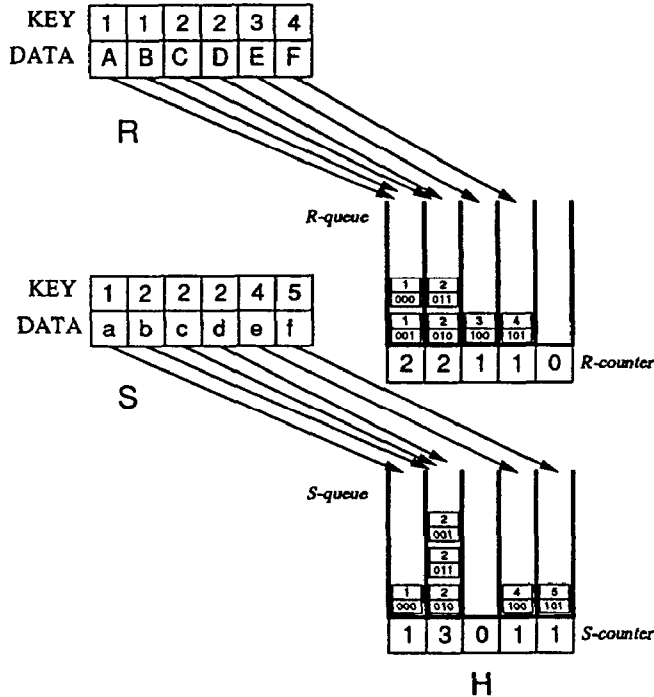


Figure10. Dynamic Clustering of Relations R and S

i) In the space H, which is allocated for the clustering, each PE corresponds to each possible hash-value and is provided with a pair of queue for two relations. One queue to receive messages from R is called *R-queue*, the other to receive message from S is called *S-queue*. The queue can pile up received messages in it. So no matter how many messages are sent to the same queue, all messages can reach

destination and be stored. Also, queues have counters which indicate how many messages are delivered to them. Two counters are called *R-counter* and *S-counter* respectively.

ii) Each tuple of relation R calculates the hash-value using its key attribute as input. Then according to hash-value, each tuple send its key-value and its own address to *R-queue* in H. Each tuple of relation S also calculates hash-value and send a message to *S-queue* in H.

Now, two queues and two counters are setup (Figure 10).

2. Local Nested Loop Join Phase

Second phase of Data Parallel Hash Join is the local join operation within each cluster, i.e., PE. We can choose any join method to perform this phase. We adopt one of the simplest algorithms, called Nested Loop Join, because it's expected that the number of tuples belonging to each cluster is sufficiently small. Here we don't consider the case for queue overflow, which is our future topics. The algorithm of Nested Loop is defined as below.

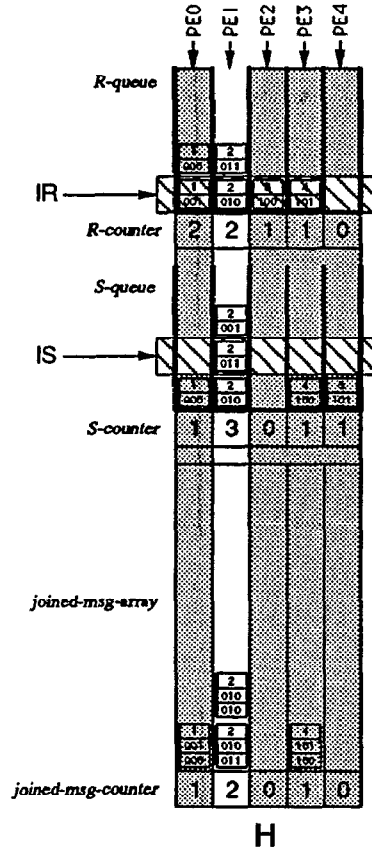


Figure11. Nested Loop Join processing over H

i) We use scalar variables named IR and IS. They are initialized by zero. we also prepare the array

joined-msg-array and its counter named *joined-msg-counter*, and initialize *joined-msg-counter* by zero.

ii) IR is compared with *R-counter*. Only PEs whose *R-counter* is larger than IR are activated. Following process is executed by only active PEs. At this point, if no cluster is active in H, Nested Loop phase is terminated.

iii) IS is initialized by zero.

iv) The clusters whose *S-counter* is equal or smaller than IS are deactivated. In example of Figure 11, only PE1 remains active because *R-counter* is larger than IR and *S-counter* is larger than IS. At this point, if no cluster is active in H, process is repeated from ii) after IR is incremented.

v) Key attribute of IRth message of *R-queue* is compared with ISth message of *S-queue*. The clusters in which two values are matched perform following operation.

Two messages used in the comparison are combined. The combined message consists of key attribute and two addresses. This is stored in the slot of *joined-msg-array* indicated by *joined-msg-counter* by using indirect addressing. The value of *joined-msg-counter* is incremented.

vi) IS is incremented and then process is jumped to iv.

3. Joined Relation Generation Phase

At this point, the numbers of messages of clusters are different, so we must even up the messages so that single PE have single message. In addition, messages in *joined-msg-array* include only addresses of tuples, so we must retrieve tuple body from R and S. In the following, we show how the final relation J is generated.

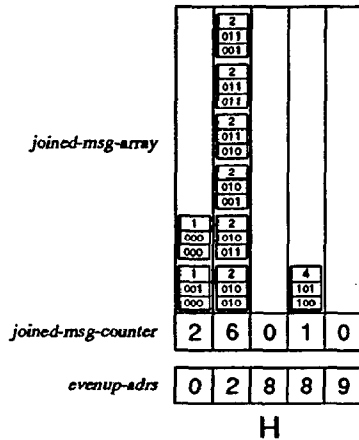


Figure12. Calculation of evenup address

i) The total sum of the values of *joined-msg-counter* is calculated. On the basis of this value, the space for J is allocated.

ii) The exclusive cumulative sums of *joined-msg-*

counter are calculated and stored in parallel variable *evenup-adrs* (Figure 12).

iii) Scalar variable I is initialized by zero.

iv) Only the clusters whose *joined-msg-counter* is greater than I are activated. At this point, if no cluster is active in H, jump to vii).

v) The active clusters send I-th element of *joined-msg-array* to J according to *evenup-adrs* (Figure 13).

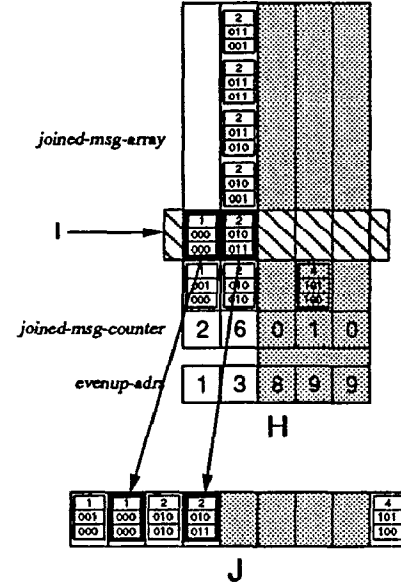


Figure13. Generation of Joined relation J

vi) The active clusters increment *evenup-adrs*. I is incremented and process is repeated from iv).

vii) Each PE of J get data from R according to the corresponding address in the message sent (Figure 14).

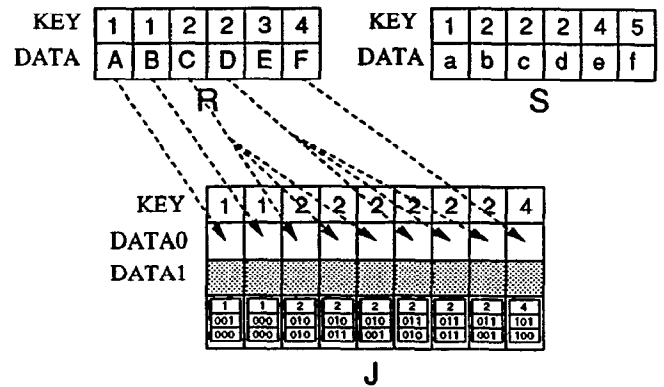


Figure14. Data bodies are derived from R and S

viii) Each PE of J get data from S according to the corresponding address in the message sent.

Thus through the above three phases, final joined relation is derived as J at Figure 9, in data parallel

fashion.

5.2 Implementation

In this section, we describe how the algorithm of Data Parallel Hash Join is implemented on the Connection Machine.

As in the case of Data Parallel Sort Merge Join, each relation R, S, J and the clustering space H are represented as VPset, and each parallel variables are represented as Field.

In the following, we describe implementation details on each phase.

1. Parallel Dynamic Clustering Phase

As well as the Data Parallel Sort Merge Join program, Data Parallel Hash Join program is made so that the sizes of VPset corresponding R, S and H can be optionally specified as VPratio such as 2, 4 or 8, independently of the numbers of tuples R and S have.

In Figure 10, each row in H including the pair of queues corresponds to one PE. To implement queues, the primitive **CM-send-to-queue** is used. After H allocates the array to store arrived messages, R and S send messages with this primitive. If the multiple messages collide, they are piled up in the appropriate position of the array in H. Using hashed value as destination address with **CM-send-to-queue**, the clustering is automatically performed. Each PE sends message to the appropriate queue simultaneously.

2. Local Nested Loop Join Phase

PEs of the Connection Machine have 1-bit flags called 'context-flag'. If this flag is set, a PE executes broadcasted instructions, otherwise doesn't execute. This flag is very important and used to implement the concept of active-set. Active-set may grow larger or smaller in the run-time, and only PEs belonging to active-set alter their own status.

During this phase, active-set grows smaller according to the scalar variables IR and IS. Context-flag manipulation is done by the primitives such as **CM-load-context** or **CM-logand-context**. Counting the numbers of active PEs is implemented by the **CM-global-count-context** primitive.

Each PE is associated with one cluster and locally perform Nested Loop Join in parallel. In this phase, no inter processor communication is needed.

Joined-msg-array is implemented as an array of Field as well as queues. Assignments and references to *joined-msg-array* is done by **CM-aset** and **CM-aref** primitives.

3. Joined Relation Generation Phase

To calculate the size of J, the **CM-global-add** primitive is used. All PEs associated with the clusters determine whether they should send message to J on the basis of their own internal status simultaneously. Sending messages from H to J is implemented by the **CM-send**. The **CM-get** primitive is used to implement J's getting data from R and S.

6 Performance Evaluation

We measured the performance of Data Parallel Sort Merge Join and Data Parallel Hash Join described above. Each implementation consists of one executable file of UNIX operating system running on the SUN workstation. Programs are written in C language, and are linked with C/Paris library. Performance evaluation was done with the size of relations R and S changed. For each of the relation sizes, we measured using one or two VPratio(s).

The performance is drawn on the X-Y plane. X-coordinate represents the number of tuples in relation R, which is equal to that in relation S. Y-coordinate represents the execution time in milliseconds needed to perform join operation. This time doesn't include the time required to load relation R and S from the front end to the Connection Machine, but the time only spent by the join operation executed on the Connection Machine.

Other performance evaluation conditions are summarized as below.

- the CM-2 hardware used for performance evaluation has 8,192 physical processors.
- the clock speed of the CM-2 is about 6.7MHz.
- the size of each tuple of R and S is 120bits (15bytes), consisting of 16bit-key and 104bit-data.
- each of R and S has exactly one tuple for each key-value ranging from 0 to *ntuple*-1 and tuples in each relation are shuffled to arrange in random order, where *ntuple* is the number of tuples in each relation. This pattern of join operation generates the same number of result tuples as that R or S has. This is called 100% join.
- for each program, relations with six sizes are used, that is, the sizes of each relation are 2048, 4096, 8192, 16384, 32768, and 65536. In case of 8192, 16384, and 32768, two VPratio are used in order to examine how the performance changes as the VPratio changes.
- only "Connection Machine busy time" is measured because elapsed time depends on the OS overhead of the front end machine.

- the performance is measured 20 times and averaged for each point in the figure.

In the following, we show the results of performance evaluations of Data Parallel Sort Merge Join and Data Parallel Hash Join.

6.1 Performance of Data Parallel Sort Merge Join

The result of performance evaluation of Data Parallel Sort Merge Join is shown in Figure 15. We used 1 as VPratio for the measurement of 2048 and 4096 tuples, and used 1 and 2 for 8192 tuples, 2 and 4 for 16384 tuples, 4 and 8 for 32768 tuples, 8 for 65536 tuples. That is, we use VPratio=1 for the small relations like 2K or 4K tuples, then we increase VPratio to hold the larger relations as needed. This is the same way we use real applications.

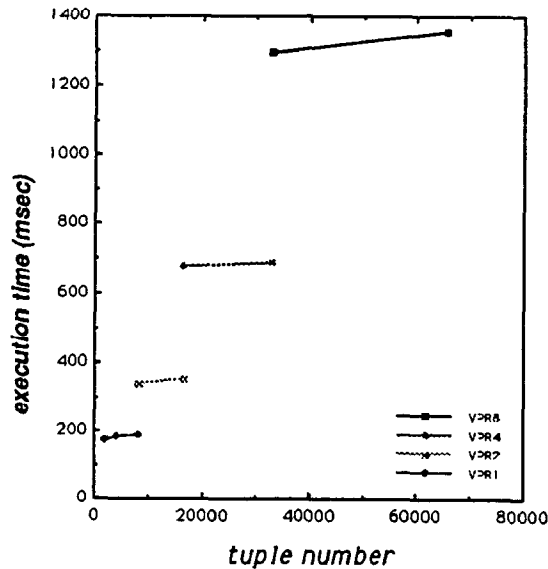


Figure15. Data Parallel Sort Merge Join Performance

In general, performance remains almost constant while the number of tuples grows, as long as VPratio remains unchanged (in other words, we can use enough number of physical processors). For example, let us examine the performance of Data Parallel Sort Merge Join when VPratio is 1 (left 3 plots in Figure 15). In this case, the time needed to perform join increases from 172msec to 187msec, that is, about 8.7% increase of processing time, while the tuple number grows 4 times larger (from 2048 to 8192). By the most straightforward solution in the serial machine, the processing time grows in proportion to the square of the number of tuples. The advantage to apply massively parallel machine to join operation is obvious.

Massively parallel machine such as the Connec-

tion Machine executes those operations such as local adding or local data moving in the exactly constant time. So one might expect that the time needed to process join operation is also constant. But it's not the case because the inter processor communications are not performed in the constant time independently of the number of PEs participating. The more PEs send or get the messages, the more time is needed to perform the inter processor communications due to hypercube congestion. We guess that the 8.7% increase of the time (8.0% degradation of the performance) is caused by this reason.

6.2 Performance of Data Parallel Hash Join

The result of performance evaluation of Data Parallel Hash Join is shown in Figure 16.

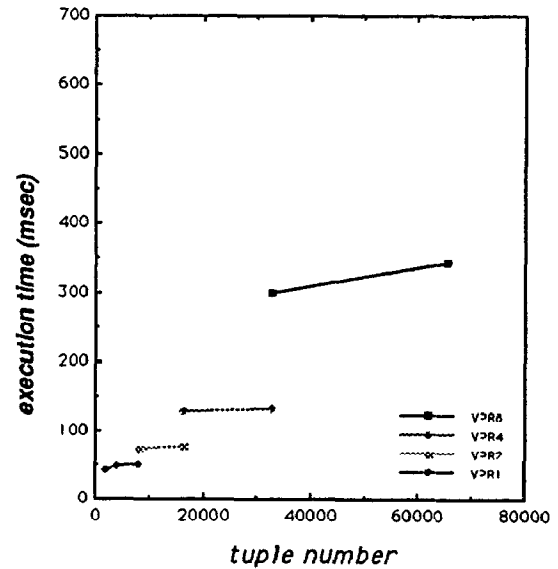


Figure16. Data Parallel Hash Join Performance

As well as Data Parallel Sort Merge Join, the execution time of Data Parallel Hash Join is almost constant while the cardinalities of relations grows larger, as long as VPratio remains unchanged.

When we think about VPratio, it may be recognized that the processing time doesn't grow in proportion to the VPratio. To think straightforward, the time is to be proportional to the VPratio because the bulk of task that each physical processor must do is proportional to the VPratio. We guess this is because some part of message passing is done within each physical processor when VPratio is high.

7 Conclusion

Massively parallel fine grain SIMD machine attracts strong attention for the future super parallel com-

puter. In this paper, we examined the effectiveness of such massively parallel machine for the relational database processing. Since join is the most important and also most time consuming operation among the relational database operations, we investigate how to parallelize the join operation on the SIMD Machine.

Most of current DBMS adopts sort merge or hash algorithm for join processing. We extend these two algorithms to fit in data parallel environment. In this paper, we described Data Parallel Sort Merge Join and Data Parallel Hash Join in detail. We clarified which portions of the algorithm steps are fully parallelized. In addition to the algorithm design, we actually implemented the system on the CM-2 and measured its performance. It is shown that join operation, which is very heavy in conventional serial machine, can be performed efficiently in massively parallel SIMD machine. $8K \times 8K$ tuple join can be executed only in less than 200 milliseconds.

In its experiments, we measured only small relation which fits the Connection Machine memory, where the execution time is almost constant with the change of the cardinality of the relation. To expect the performance for larger relation, we measured the performance changing the VPratio. We found that the performance degradation is less than linear.

However, if we are to handle much larger relation, we have to use secondary storage system, which is called DataVault in the CM-2. We plan to measure the performance of the relational operation with very large relation residing in DataVault. We would like to run $10M\text{-tuple} \times 10M\text{-tuple}$ Wisconsin Join ($1G\text{bytes} \times 1G\text{bytes}$ Join) using GRACE hash and Hybrid Hash algorithm.

Acknowledgment

Dr. Craig Stanfill and Dr. Daniel Hillis gave us very valuable suggestion for this research. We extend our gratitude to Thinking Machines Corporation and TEPCO for their kind support.

References

- [1] E. Ozkaran: Database Machines and Database Management, Prentice-Hall, N.J., U.S.A., 1986.
- [2] S. Y. W. Su: Database Computers, McGraw-Hill, N.Y., U.S.A., 1988.
- [3] M. Kitsuregawa, *et al*: Design and implementation of high speed pipeline merge sorter with run length tuning mechanism, Database Machines and Knowledge Base Machines, Kluwer Academic publishers, Kitsuregawa (ed.), 1988.

- [4] H. Schweppe, H. C. Zeidler, W. Hell, H. O. Leilich, G. Stiege, and W. Teich: RDBM - A Dedicated Multiprocessor System for Database Management, in Advanced Database Machine Architecture, D. K. Hsiao (ed.), Prentice-Hall, Englewood Cliffs, N.J., pp.36-86, 1983.
- [5] K. Kojima, *et al*: IDP - A Main Storage Based Vector Database Processor, Database Machines and Knowledge Base Machines, Kluwer Academic Publishers, Kitsuregawa Ed. 1988.
- [6] M. Kitsuregawa, H. Tanaka, and T. Moto-oka: Application of Hash to Data Base Machine and Its Architecture, New Generation Computing, Vol.1, No.1, pp.66-74, 1983.
- [7] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood: implementation Techniques for Main Memory Database Systems, in ACM SIGMOD '84, pp.1-8, 1984.
- [8] M. Kitsuregawa, *et al*: Functional Disk System for Relational Database, Proc. of the 3rd Int. Conf. on Data Engineering, pp.88-95, 1987.
- [9] M. Kitsuregawa, *et al*: Query Execution for Large Relations on Functional Disk System, Proc. of Int. Conf. on Data Engineering, pp.159-167, 1989.
- [10] D. J. DeWitt, *et al*: GAMMA - A High Performance Dataflow Database Machine, Proc. of Int. Conf. on Very Large Data Bases, pp.228-237, 1986.
- [11] D. J. DeWitt, *et al*: A Performance Evaluation of Four Parallel join Algorithms in a Shared-Nothing Multiprocessor Environment, Proc. of SIGMOD, 1989.
- [12] Teradata Corp.: DBC/1012 Data Base Computer Concepts and Facilities, CO2-0001-05.
- [13] C. Stanfill: personal communication.
- [14] Thinking Machines Corp.: Connection Machine Model CM-2 Technical Summary, Version 5.1, 1989.