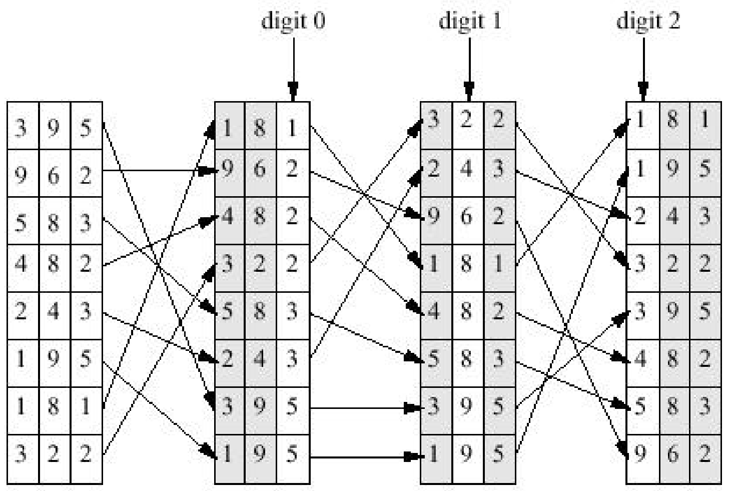
# Radix Sort and Hash-Join for Vector Computers

Ripal Nathuji

6.893: Advanced VLSI Computer Architecture

10/12/00

## What is Radix Sorting?



| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 9 5 | | 1 8 1 | | 3 2 2 | | 1 8 1 |
| 9 6 2 | | 9 6 2 | | 2 4 3 | | 1 9 5 |
| 5 8 3 | | 4 8 2 | | 9 6 2 | | 2 4 3 |
| 4 8 2 | | 3 2 2 | | 1 8 1 | | 3 2 2 |
| 2 4 3 | | 5 8 3 | | 4 8 2 | | 3 9 5 |
| 1 9 5 | | 2 4 3 | | 5 8 3 | | 4 8 2 |
| 1 8 1 | | 3 9 5 | | 3 9 5 | | 5 8 3 |
| 3 2 2 | | 1 9 5 | | 1 9 5 | | 9 6 2 |

- Sort by least significant digit instead of most significant digit
- Better than sorting by most significant digit since it saves having to keep track of multiple sort jobs

## Properties of Radix Sorting Algorithms

- Treat keys as multidigit numbers, where each digit is an integer from <0…(*m*-1)> where *m* is the radix

- The radix *m* is variable, and chosen to minimize running time

  Example:

  32-bit key as 4 digit number

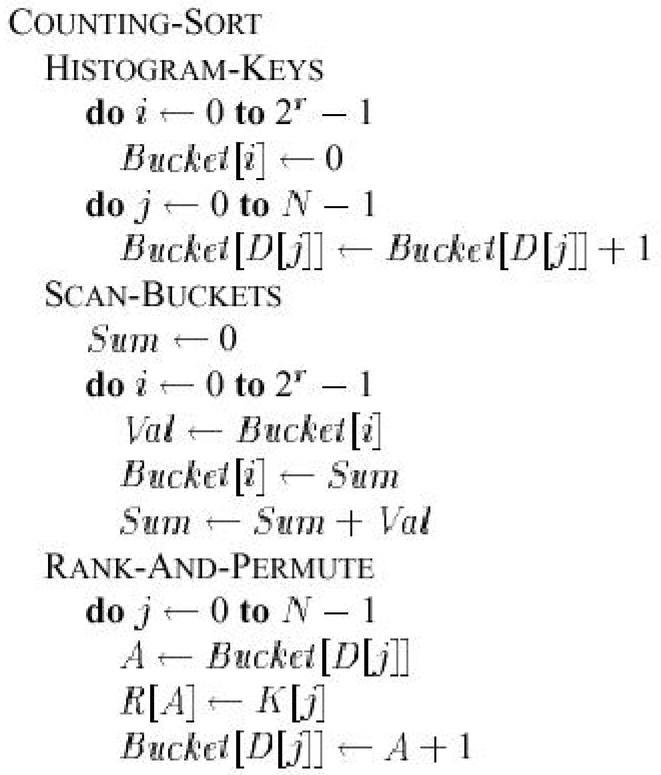
  *m* is equal to the number of distinct digits so $m = 2^{32/4} = 2^8 = 256$

- Performance: Runs in $O(n)$

  Other comparison based sorts such as quicksort run in $O(n \log n)$ time

  ***Not advantageous for machines w/cache

## Serial Radix Sort



```
COUNTING-SORT
  HISTOGRAM-KEYS
    do $i \leftarrow 0$ to $2^r - 1$
      $Bucket[i] \leftarrow 0$
    do $j \leftarrow 0$ to $N - 1$
      $Bucket[D[j]] \leftarrow Bucket[D[j]] + 1$
  SCAN-BUCKETS
    $Sum \leftarrow 0$
    do $i \leftarrow 0$ to $2^r - 1$
      $Val \leftarrow Bucket[i]$
      $Bucket[i] \leftarrow Sum$
      $Sum \leftarrow Sum + Val$
  RANK-AND-PERMUTE
    do $j \leftarrow 0$ to $N - 1$
      $A \leftarrow Bucket[D[j]]$
      $R[A] \leftarrow K[j]$
      $Bucket[D[j]] \leftarrow A + 1$
```

- N = # of keys to sort
  K = array of keys
  D = array of r-bit digits

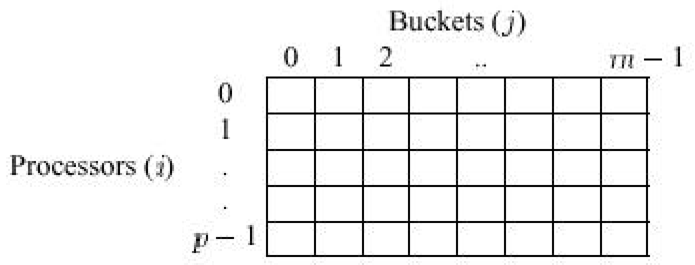
Values of Bucket[] after each phase:

- Histogram-Keys:
  Bucket[*i*] contains the number of digits having value *i*
- Scan-Buckets:
  Bucket[*i*] contains the number of digits with values < *i*
- Rank-And-Permute:
  Each key of value *i* is placed in its final location by getting the offset from Bucket[*i*] and incrementing the bucket

## How Can We Parallelize the Serial Radix Sort?

Problem:
- Loop dependencies in all three phases

Solution:
- Use a separate set of buckets for each processor Each processor takes care of N/P keys where P is number of processors.



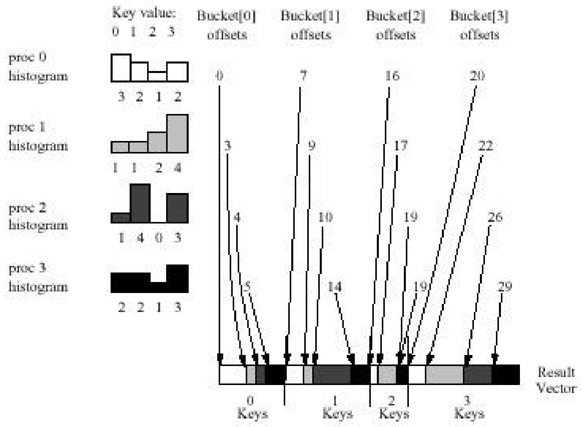
This resolves the data dependencies, but creates a new problem with Scan-Buckets: How can we sort the digits globally instead of just within the scope of each individual processor.

## Fully Parallelizing Scan-Buckets

Instead of having each processor simply scan its own buckets, after doing Scan-Buckets we would like the value of Buckets[*i*,*j*] to be:

$$\sum_{k=0}^{p-1} \sum_{m=0}^{j-1} Buckets[k,m] + \sum_{k=0}^{i-1} Buckets[k,j]$$

The sum can be calculated by flattening the matrix and executing a Scan-Buckets on the flattened matrix

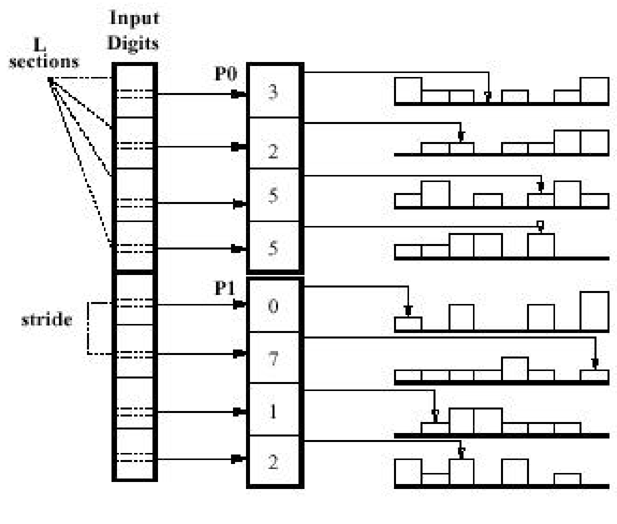


## Techniques Used In the Data-Parallel Radix Sort

- Virtual Processors
- Loop Rakings
- Processor Memory Layout

## Virtual Processors

- Vector multiprocessors offer two levels of parallelism: multiprocessor facilities and vector facilities.
- To take advantage of this, view each element of a vector register as a virtual processor. So a machine with register length L and P processors has L x P virtual processors.
- Now the total number of keys can be divided into L x P sets.

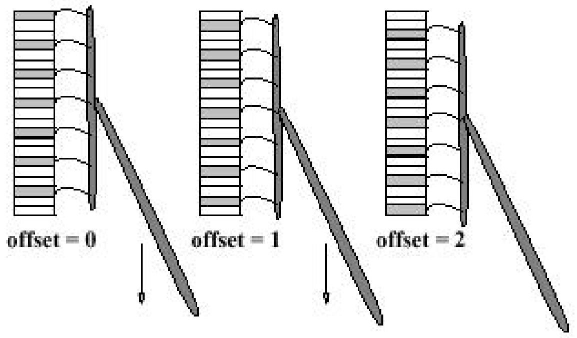


## Loop Raking

- Usually operations on arrays are vectorized using strip mining. In strip mining an element of a vector register handles every Lth-element

- Unfortunately using strip mining each virtual processor will have to handle a strided set of keys instead of a contiguous block as required by the parallel algorithm

- Using a technique called loop raking, each virtual processor handles a contiguous block of keys. Loop raking uses a constant stride of N/L to access elements



## Processor Memory Layout

- A memory location X is contained in bank (X mod B) where B is the number of banks
- Simultaneous accesses to the same bank result in delay

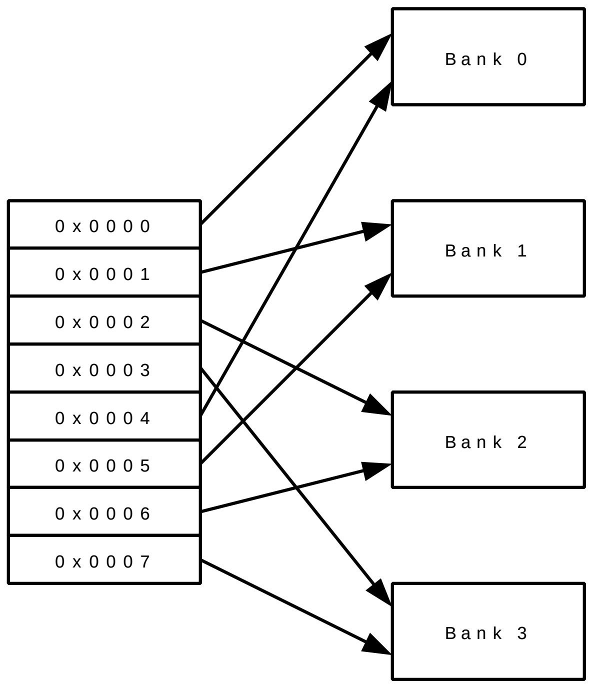
There are two possible ways to lay out the buckets in memory:

- Place the buckets for each virtual processor in contiguous memory locations:

  This approach could cause multiple virtual processors to access the same bank simultaneously.
- Place the buckets so that the buckets used by each virtual processor are in separate memory banks (i.e. Place all the buckets of a certain value from all virtual processors in contiguous memory locations):

  This approach keeps multiple virtual processors from accessing the same bank simultaneously

# Processor Memory Layout: Example



## Implementation of Radix Sort on 8-processor CRAY Y-MP

Four Routines:

1. Extract Digit:
   - Extracts current digit from keys and computes an index into the array of buckets
   - Uses loop raking
   - Time for routine: $T_{\text{Extract-Digit}}=1.2 \cdot N/P$
2. Histogram Keys:
   - Uses loop raking
   - Time for routine: 2 steps

     $T_{\text{Clear-Buckets}}=1.1 \cdot 2^r \cdot L$

     $T_{\text{Histogram-Keys}}=2.4 \cdot N/P$
3. Scan Buckets:
   - Uses loop raking
   - Time for routing: $T_{\text{Scan-Buckets}}=2.5 \cdot 2^r \cdot L \cdot P/P= 2.5 \cdot 2^r \cdot L$

## Implementation of Radix Sort on 8-processor CRAY Y-MP

4. Permute Keys:
   - Uses loop raking
   - Time to permute a vector ranges from 1.3 cycles/element to 5.5 cycles/element
   - Time for routine: $T_{\text{Rank-And-Permute}}=3.5\cdot N/p$

## Performance Analysis

Total sorting times:

- $T_{\text{Counting-Sort}}=L\cdot 2^{r}\cdot T_{\text{bucket}}+N/P\cdot T_{\text{key}}$
- $T_{\text{Radix-Sort}}=b/r(L\cdot 2^{r}\cdot T_{\text{bucket}}+N/P\cdot T_{\text{key)}}$

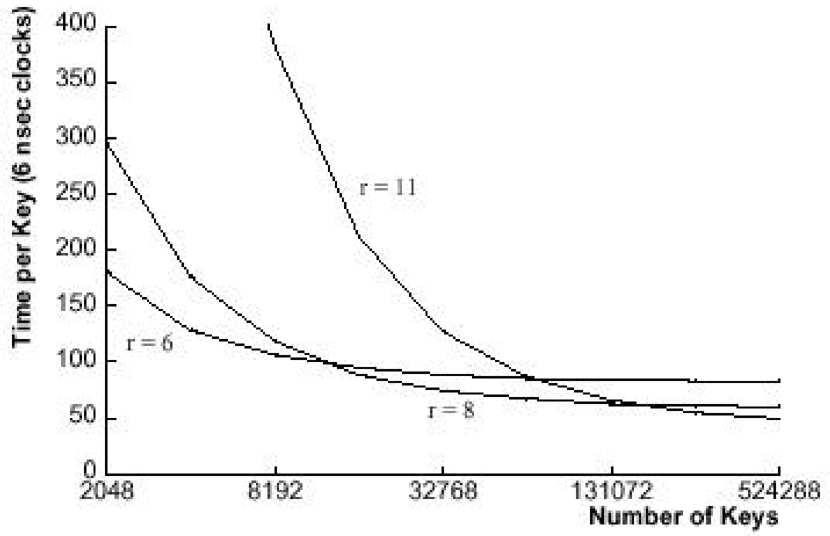
Choice of Radix:

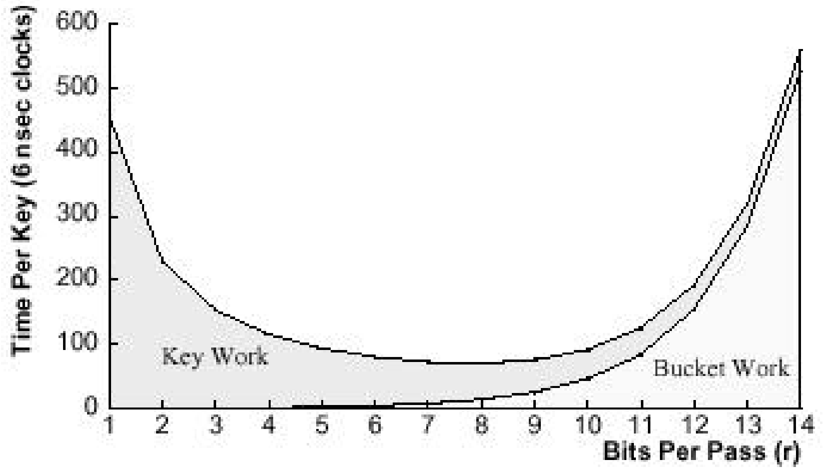
- The optimal value for *r* increases with the number of elements per processor
- Choosing *r* below the optimal value puts too much work on keys, choosing *r* above the optimal value puts too much work on buckets
- Value for *r* and approximation of total sort time:

$$\begin{aligned} r &= \lg\left(N/P\cdot\frac{T_{\text{key}}}{L\cdot T_{\text{bucket}}}\right)-\lg(r\ln 2-1) \\ &\approx \lg(N/P)-\lg(L)-1 \end{aligned}$$

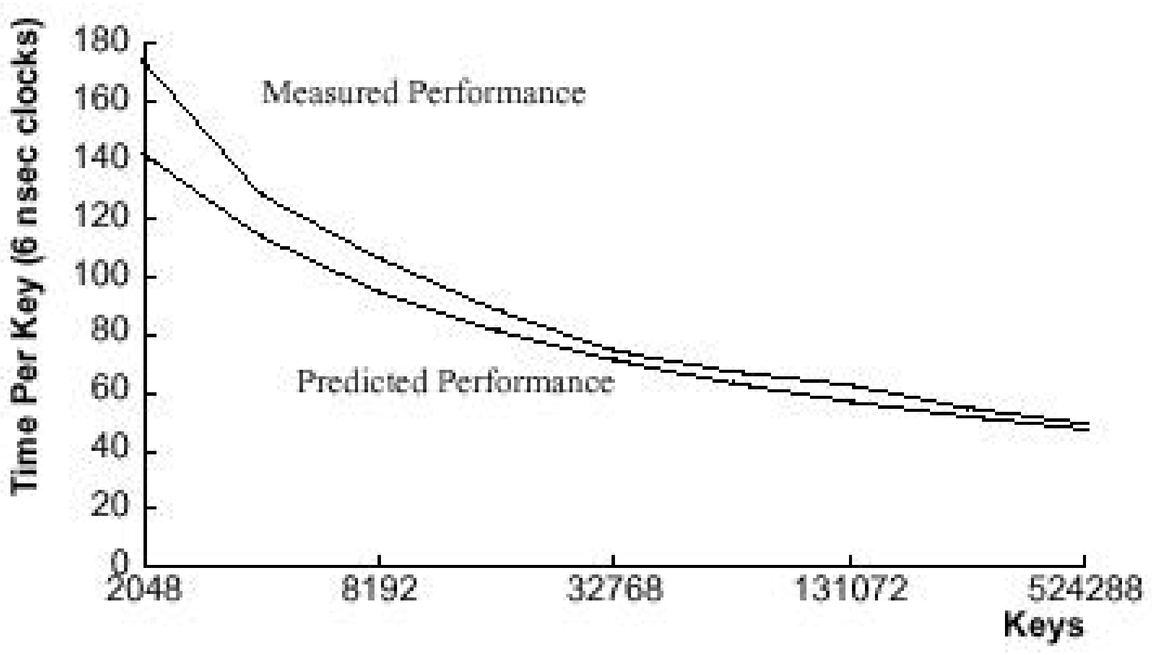
$$T_{\text{Radix-Sort}}\approx\frac{b\cdot(N/P)}{\lg(N/P)-\lg(L)-1}\left(\frac{T_{\text{bucket}}}{2}+T_{\text{key}}\right)$$

## Choosing a Value for *r*





## Predicted vs. Measured Performance



## Other Factors of Performance

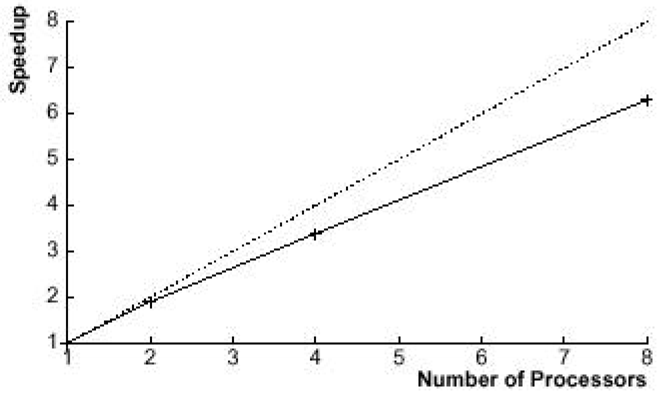
- Vector Length
- Multiple Processors
- Space

## Varying Vector Length

- Decreasing the vector length decreases the number of virtual processors
- Advantage: decreases the time for cleaning and scanning buckets
- Disadvantage: increases the cost per element for performing the histogram, $T_{key}$
- Conclusion: Reducing the vector length is only beneficial if $(N/P < 9000)$

## Change in Performance with Number of Processors

- If N/P is held constant, speedup is linear with increase in P



- If N is fixed, speedup is not linear with increase in P due to changes in the optimal *r*

## Memory Issues

Memory needed for Radix Sort:

- Temporary array of size N to extract current digit + an array of size N for destination of permute + array of size $L \cdot 2^r \cdot P$ for the buckets $\approx 2.5N$

Possible ways to conserve memory:

- Extract digit as needed instead of using temporary array
- Lower radix (i.e. $2^r$ term)
- Reduce vector length (L)

## Conclusions on Vectorized Radix Sort

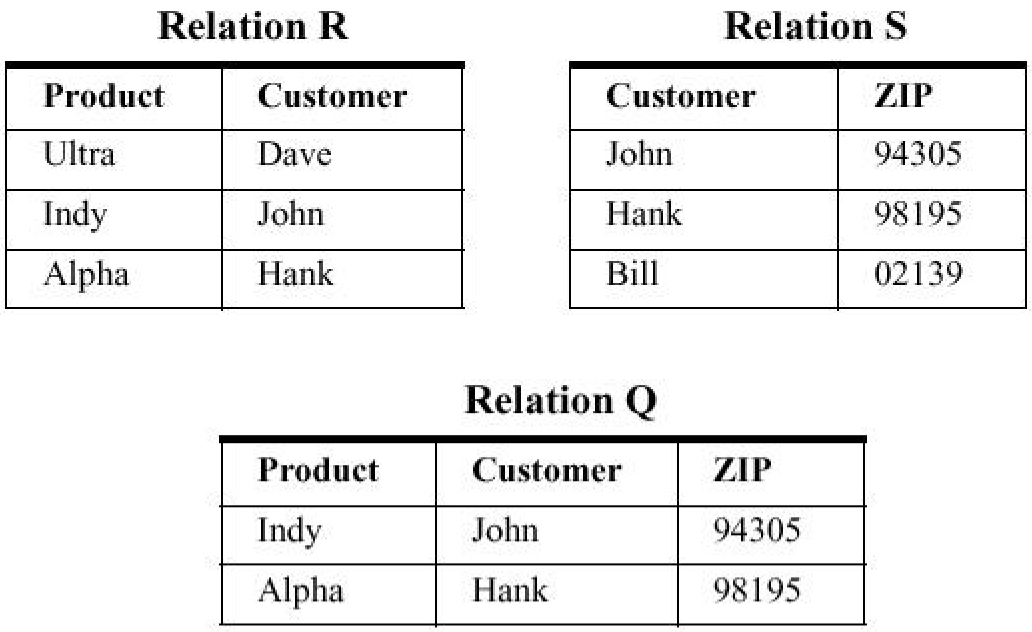
- Radix sort can be Vectorized using three major techniques
    1. Virtual processors
    2. Loop raking
    3. Efficient memory allocation

- Overall performance can be optimized by adjusting
    1. The radix *r*
    2. The vector length L
    3. Number of processors
    4. Memory considerations

## Introduction to Hash-Join

- The join operation is one of the most time-consuming and data-intensive operations performed in databases
- The join operation is frequently executed and used
- Idea: vectorize the computational aspects of the hash and join phases

## Equijoin



**Relation R**

| Product | Customer |
|---|---|
| Ultra | Dave |
| Indy | John |
| Alpha | Hank |

**Relation S**

| Customer | ZIP |
|---|---|
| John | 94305 |
| Hank | 98195 |
| Bill | 02139 |

**Relation Q**

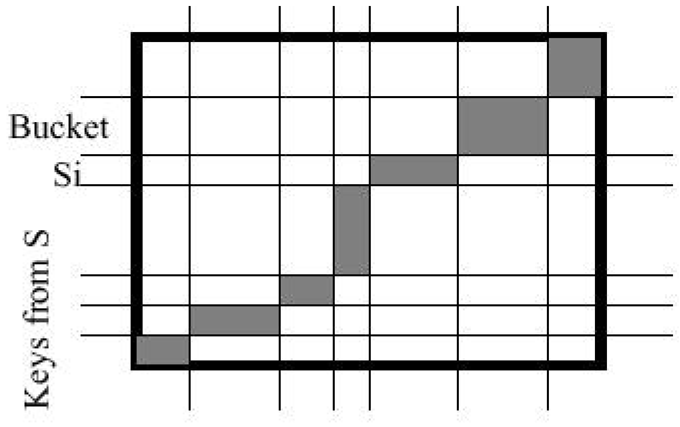
| Product | Customer | ZIP |
|---|---|---|
| Indy | John | 94305 |
| Alpha | Hank | 98195 |

## Naive Approach

```
For each row in S do
    For each row in R do
        if (Ra.key == Sb.key) then
                concatenate Ra, Sb and place in result Q
```

This approach is too expensive and runs in $O(n^2)$

## Reduction of Loads by Hashing



By hashing the tuples of each relation into buckets, we change from having to compare the entire area to just the areas in which keys hash to the same bucket (shaded areas).

## Grace-Hash Join

Two Phases:

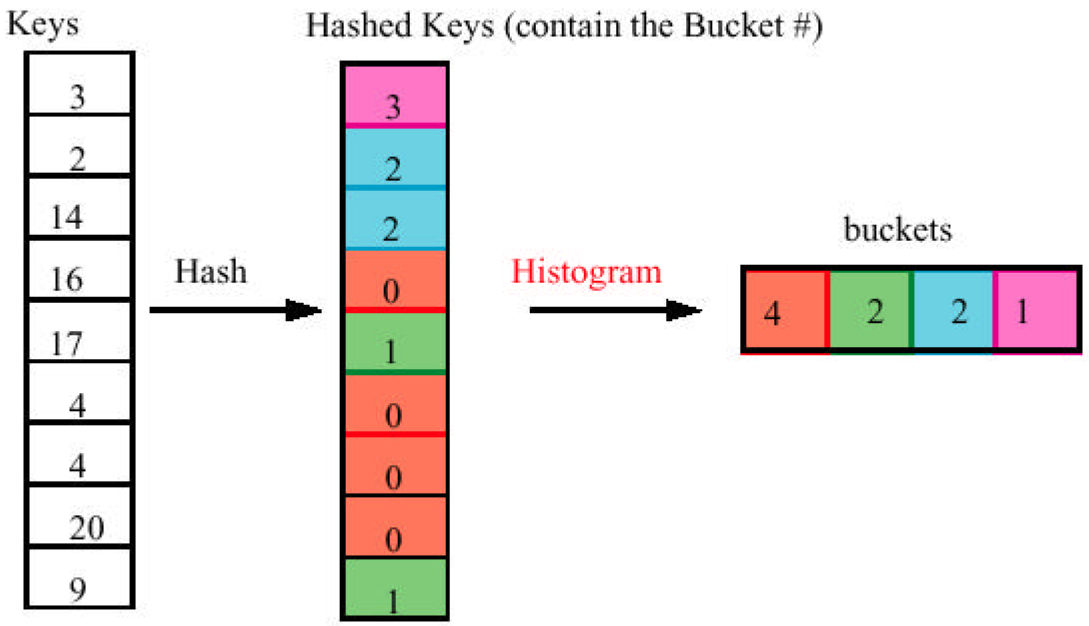
1. Relations are hashed into buckets so that each bucket is small enough to fit into main memory

2. A bucket from one relation is brought into memory and hashed. Then every key of the second relation is hashed and compared to ever key of the first relation which hashed to the same bucket.

## Phases of Sequential Hash

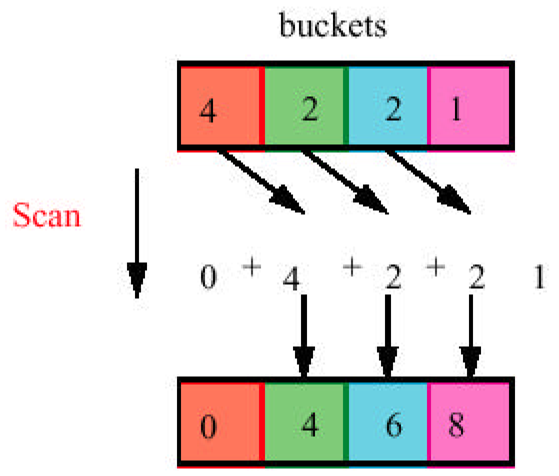
- Extract-Buckets and Histogram Phases
- Scan Phase
- Rank and Permute Phase

## Extract-Buckets and Histogram Phase



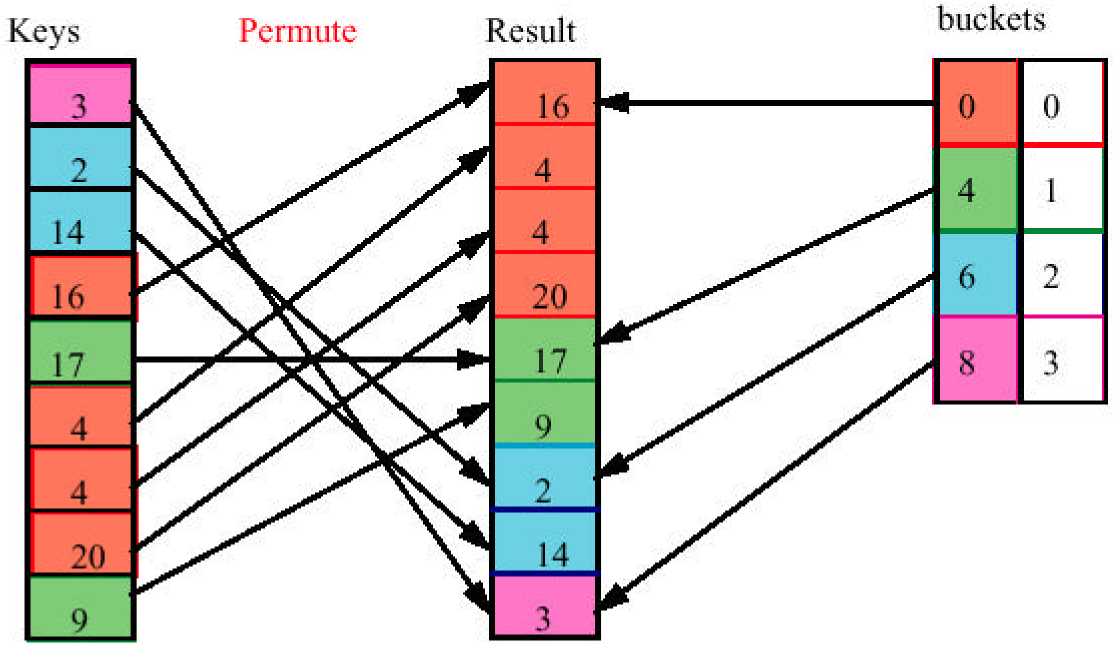
Hash function used is key *mod* (number of buckets)

## Scan Phase



```
sum :=0;
for (i :=0 to number of buckets ) {
        val := buckets[i];
        buckets[i] := sum;
        sum += val;
}
```

## Rank and Permute Phase



After this phase the result and buckets arrays form a hash table

## Sequential Join Algorithm

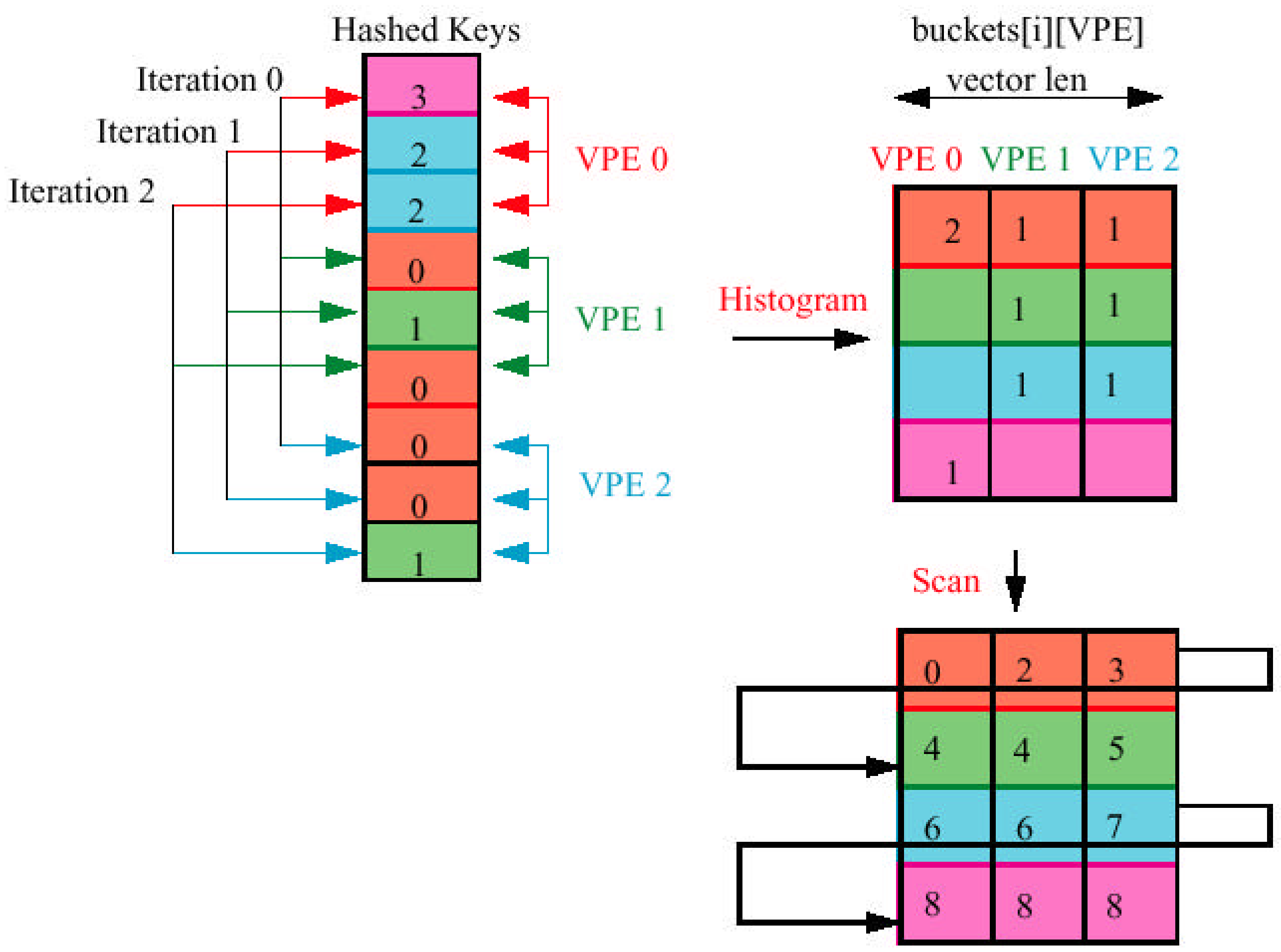
- The disk bucket $R_i$ is brought into memory
- Each record of $S_i$ is hashed and compared to every record in $R_i$
- Any matches that are found are concatenated and written to final output file

## Vectorized Algorithm

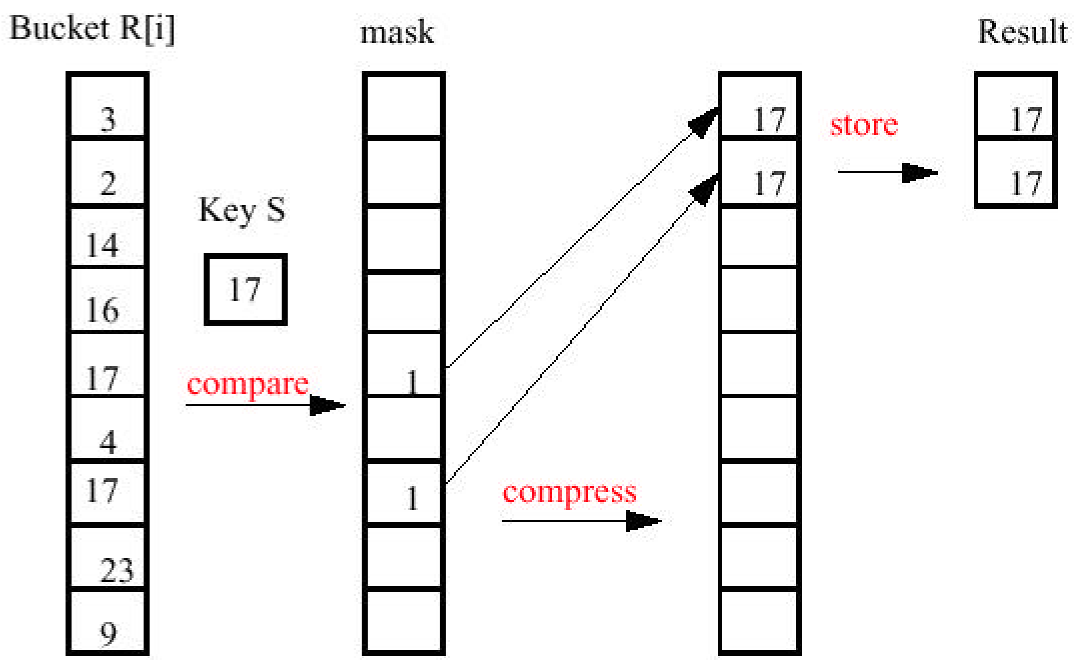
Use two techniques:

1. Virtual processors

2. Loop raking


Hashed Keys
Iteration 0
Iteration 1
Iteration 2
3
2
2
0
1
0
0
0
1
VPE 0
VPE 1
VPE 2
Histogram
buckets[i][VPE]
vector len
VPE 0 VPE 1 VPE 2
2 1 1
1 1
1 1
1
Scan
0 2 3
4 4 5
6 6 7
8 8 8

# Join



Mask vector is generated by a scalar-vector comparison

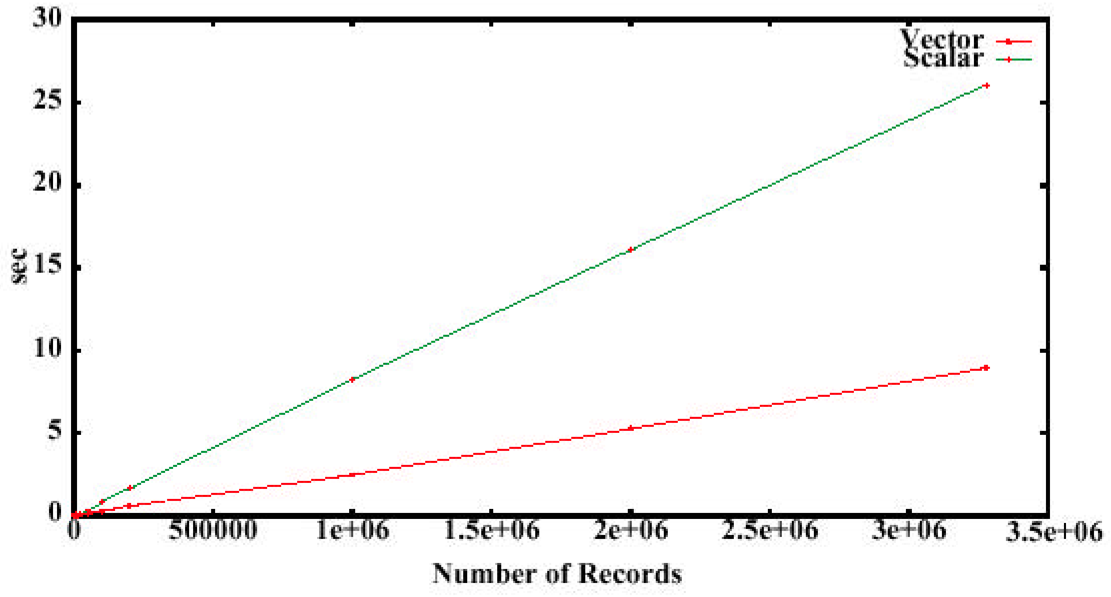
## Problems that Occurred

Compiler exhibited problems vectorizing certain parts of the code:

- Getting the compiler to vectorize certain loops in the code

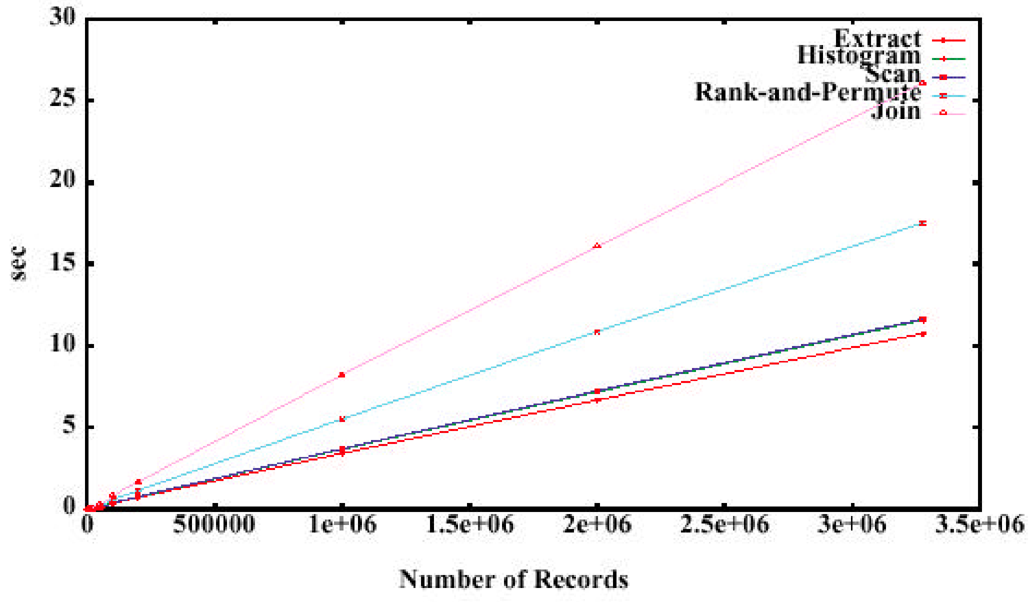
- The compiler would not vectorize compress in the Join phase

## Results (using CRAY C90)

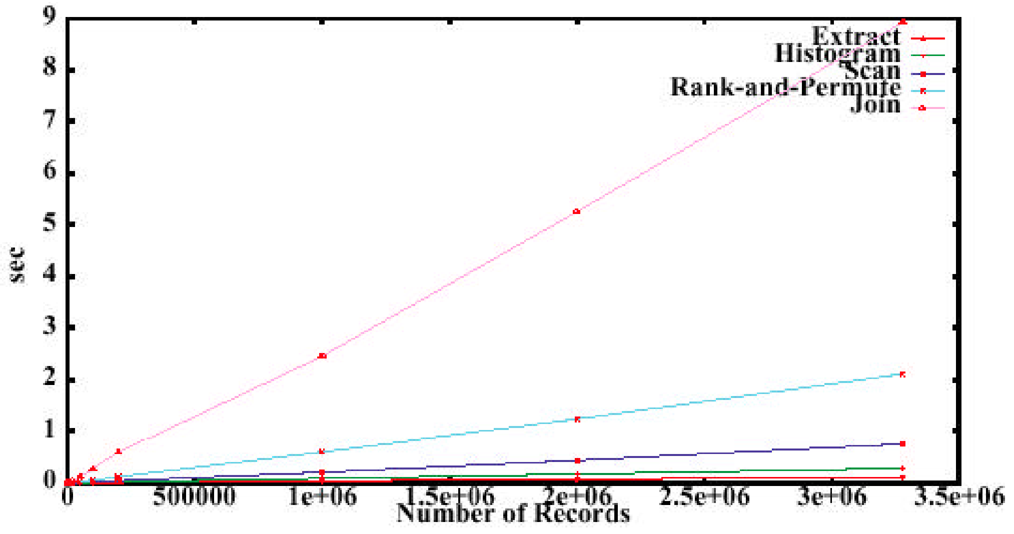


## Results

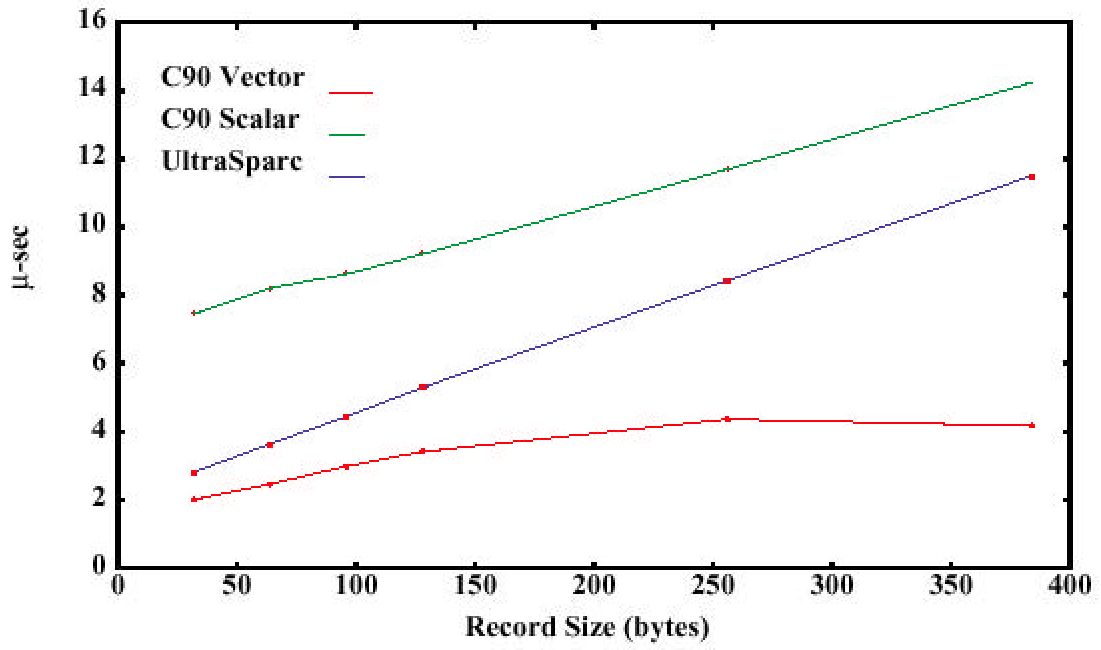
Scalar:



Vector:



# Results


C90 Vector
C90 Scalar
UltraSparc
µ-sec
Record Size (bytes)
0
2
4
6
8
10
12
14
16
0
50
100
150
200
250
300
350
400