# A Method to Implement Equijoins with MPI and OpenMP

# Abstract

The speed of processors does not double every 18 to 24 months like Moore’s law predicted anymore. Parallel programming has therefore become even more important as a means of increasing processing speed and has found numerous applications in industry. This paper will focus on its application in database query processing, specifically, equijoins between relational database tables. Specifically, it will examine how using a hybrid model of MPI and OpenMP compares with using OpenMP only.

# Problem Description and Introduction

Let the tables be joined be represented R1 and R2 and the table resulting from joining them by R3. R1 has the columns A and B; R2 has the columns A and C. The tables to be joined will be stored in two CSV files. Each CSV file will have the following format

key 1, key 2

key 1, key 2

.

.

.

Key 1, key 2

Where for each row, key 1 is a key from column A (the common column) and key 2 is from column B or C depending on the file.

# Solution and Implementation

**MPI+OpenMP Equijoin Implementation**

The master MPI process (node 0) will read the two CSV files into tables in the C program for joining them. A table is represented in by two arrays each representing one column of the table. This representation of tables helps avoid the extra complexity involved in managing a multidimensional array representation. A tuple will therefore be represented by two elements from the two arrays for a table that share the same index. Tables R1 and R2 are sent to MPI nodes 1 and 2 respectively.

|  |  |
| --- | --- |
| **A** | **B** |
| Item 1 | Item 1 |
| Item 2 | Item 2 |
|  |  |
|  |  |
|  |  |
| Item n | Item n |

A1 = [item 1, item 2, …, item n]

B = [item 1, item 2, .., item n]

**Equivalent Representations of R1**

Processing within node 1

Node 1 receives R1’s two arrays from node 0. It sorts both arrays based on array 1 using the Merge Sort algorithm. This means that array B is rearranged such that if key x in B occupied the same index as key y in A, its position after sorting will be the new position of key x. Why sorting is done will discussed later be later. After sorting, a bloom filter is applied to array A. The hash functions used in the bloom filter division method, the mid square method and digit folding method. The resulting bit string is sent to node 2 and waits for a reply. The reply received contain two arrays holding tuples from R2 that may qualify for the join. Let’s call the two arrays A2 and C (they are also sorted like A1 and B).

A binary search is done for each key in A1 in A2 in order to locate a tuple that qualifies for the join. It is here that sorting proves advantageous. Since there can be repetitions of keys, having the arrays unsorted will mean that if a key k is repeated ten times, ten searches will have to be conducted. It will also mean that a binary search cannot be used leaving the less efficient linear search. Sorting ensures that tuples with the same values for A appear consecutively. This makes it simple to obtain all tuples that share the same key value for A. Checking the indexes around a key value will yield all keys that share the same value. All possible combinations of tuples that qualify can then be computed. Another advantage of sorting also produces a table R3 that is can is sorted on column A and can easily be searched on that column.

**Algorithm for MPI and OpenMP Implementation**

**ALGORITHM** mpiJoin(fileA, fileB, fileC)

//Reads two CSV files holding relations, conducts and equijoin on them and writes the

//resulting relation to an output file

//Input: names of two CSV files holding relations fileA and fileB

//Output: a CSV file named fileC holding a joined table

A1 ← array of values in first column of fileA

B ← array of values in second column of fileA

A2 ← array of values in first column for fileB

C ← array of values in first column for fileB

Sort A1 and B with MergeSort based on A1 using OpenMP

Sort A2 and C with MergeSort based on A2 using OpenMP

Initialize the 3 MPI processes node 0, node 1 and node 2

**//Steps carried out in node 0**

Send A1 from node 0 to node 1

Send B from node 0 to node 1

Send A2 from node 0 to node 2

Send C from node 0 to node 2

**//Steps carried out in node 1**

Receive A1 from node 0

Receive B from node 0

F ← array of bits

l ← length of A1

for i ← 0 to l do

m ← h1(A1[i])

F[m] ← 1

n ← h2(A1[i])

F[n] ← 1

p ← h3(A1[i])

F[p] ← 1

Send F from node 1 to node 2

Receive newA2

Receive newC

pos ← 0

start ← 0

end ← 0

index ← 0

finalA ← empty array // to hold final A column

finalB ← empty array // to hold final B column

finalC ← empty array // to hold final C column

for i ← 0 to l do

if (i = 0) or (A[i] != A[i-1])

// binarySearch(a, b) searches for value a in array b

pos ← binarySearch(A[i], newA2)

start ← pos

end ← pos

if pos != -1

decrease start until it equals the index of the first instance of A[i] in newA2

increase end until it equals the index of the last instance of A[i] in newA2

if pos != -1

for j ← start to end do

finalA[index] ← A1[i]

finalB[index] ← B[i]

finalC ← newC[j]

index ← index + 1

Write finalA, finalB and finalC to fileC

**//Steps carried out in node 2**

Receive A2 from node 0

Receive C from node 0

Receive F from node 1

t ← length of A2

index ← 0

newA2 ← empty array // to hold values from A2 which may qualify for a join

newC ← empty array // to hold values from A2 which may qualify for a join

for i ← 0 to t do

m ← h1(A2[i])

n ← h2(A2[i])

p ← h3(A2[i])

if F[m] = 1 and F[n] = 1 and F[p] = 1

newA2[index] ← A2[i]

index ← index + 1;

Send newA2 to node 1

Send newC to node 1

**OpenMp Only Equijoin Implementation**

This implementation of the join here will be done using n OpenMP threads. Like the MPI+OpenMP implementation, this implementation will start with sequential I/O to read the CSV files R1 and R2 into the program and sort the resulting arrays with multiple OpenMP threads. Since R1 and R2 would not be on different nodes, it will not be necessary to apply a bloom filter, rather the program can go straight ahead with the join in very much the same way it does in the MPI+OpenMP system.

**Algorithm for OpenMP Implementation**

**ALGORITHM** mpiJoin(fileA, fileB, fileC)

Spawn n OpenMP threads

A1 ← shared array of values in first column of fileA

B ← shared array of values in second column of fileA

A2 ← shared array of values in first column for fileB

C ← shared array of values in first column for fileB

Sort A1 and B with MergeSort based on A1 using

Sort A2 and C with MergeSort based on A2 using OpenMP

pos ← 0

start ← 0

end ← 0

index ← 0

finalA ← empty array // to hold final A column

finalB ← empty array // to hold final B column

finalC ← empty array // to hold final C column

for i ← 0 to l do

if (i = 0) or (A[i] != A[i-1])

// binarySearch(a, b) searches for value a in array b

pos ← binarySearch(A[i], A2)

start ← pos

end ← pos

if pos != -1

decrease start until it equals the index of the first instance of A[i] in A2

increase end until it equals the index of the last instance of A[i] in A2

if pos != -1

for j ← start to end do

finalA[index] ← A1[i]

finalB[index] ← B[i]

finalC ← newC[j]

index ← index + 1

Write finalA, finalB and finalC to fileC

# Results and Conclusion

Due to bugs in our code, we were unable to evaluate the performance of our implementation against MapReduce as initially intended. Rather we will logically compare the two.

If we assume that task parallelized in the OpenMP only system is also parallelized in the OpenMP+MPI system, the OpenMP only system should be superior as extra time is not spent sending data between threads. However, this is only the case if the OpenMP system is running on a system with sufficient storage. Once storage proves insufficient, the MPI+OpenMP system becomes the better option.