Parallel Blocking Nested Loop Join: Intel Phi Co-Processor Environment

Zhila Nouri, Alireza Chakeri, Rahul Padmanabhan

*Abstract*— It is of common knowledge that the joins operator is one of the most commonly used as well as expensive operators in SQL. Although a Cartesian product is never calculated during the performance of a joins function due to the obvious reason of excessive data evaluation with the high probability of a system crash, other bottlenecks arise. Joins are particularly expensive due to issues such as bulk disk reads and sequential processing. In our day to day lives, in the database computing world, joins are commonly used across several tables together. This considerably reduces the performance of a join as the volume of the data in the tables increase. It is thus essential to have an efficient method for the execution of the joins operator. A method we can use to provide a performance enhancement to the join operation is to employ the use of parallel processing. Also hardware development has rapidly far surpassed the programming ability of the current day by the introduction of processors with multiple cores. Multiple cores enable us to process instructions in parallel, and thereby arrive at the result in a faster duration of time compared to the instructions being processed sequentially. In this project, we propose the implementation of blocking nested loop join operator in a parallel processing environment on the Intel Phi co-processor. Particularly, one table is partitioned into several blocks according to cache size of cores and each block is putted into cash of each cores. Also, the other table is divided into several chunks and each chunk handles by one thread. The propose approach yields a performance gain of a highly expensive function.

# Introduction

It is inevitable that future’s processors will have a large number of cores. Hence, the primary problem is that, despite the large number of cores a processor has, software will not be written to optimize performance by maximum utilization of all the cores. This therefore leaves the resulted staggering hardware advancement in a state of void. For the past few years companies like Intel have been introducing processors with more than 6 cores, and the industry expects this trend to continue and increase. Unlike Moore’s law (which held true for decades), we do not have a law that predicts the exponential increase in the number of cores of a processor and all we can make are estimates at this point. In this project, we use the Intel Xeon Phi coprocessor to achieve the aim of parallel computing on the joins operator to increase efficiency. The Intel Phi coprocessor has 60 cores and is high-performance computing hardware with a threaded programming model. It employs the MIC (Many Integrated Core) architecture which combines several cores into a single chip to give developers an edge by running programming tools and methods that are present.

An SQL query can be written as an expression in relational algebra. Relational algebra can be used to express SQL queries in a procedural format. Some of such relational algebra operations are Selection, Projection, Union, Intersection, Join and Cartesian product. We are focusing on the JOIN operation in this project. There are also several types of JOIN operations in SQL such as Inner Join, Outer Join, Left Join and Right Join. The default type of join is the inner join which takes only the records from two tables based on a condition or set of conditions. Particularly, an inner join can be expressed as the result of initially taking the Cartesian product of all records in the tables to be joined combining them with the join criteria and returning only those rows. In addition, there are several joins algorithms available including Sort Merge Join, Cluster Join, Hash Join, Nested Loop Join, Block Nested Loop Join and etc.

On the other hand, as the volume of data increases, the cost required to mine information generally increases as well. It is imperative that we get the resulting data in time otherwise the usefulness of the data normally depreciates. For instance, consider a banking industry. If it takes two days to compute that a group of people are fraudulently using checks to withdraw more than the funds available in the account, the loss is already incurred without preventive measures being taken in time. Specifically, the objective of this project is to analyze the effect of an efficient parallelization approach to enhance the performance of current join algorithms. We propose a parallel blocking nested loop inner join algorithm such that one table is partitioned into several blocks according to cache size of cores and each block is putted into cash of each cores, and other table is divided into several chunks and each chunk handles by one thread of different core.

# Parallel Blocking Nested Loop Join

## Blocking Nested Loop Join

Consider two non-empty tables R and S. A basic nested loop join (NLJ) can be stated as follows:

*For each tuple r in R do*

*For each tuple s in S do*

*If r and s satisfy the join condition*

*Then output the tuple <r,s>*

As one can see, the I/O cost for the simple NLJ would be very expensive for large tables which makes it impractical in real world situations. An approach to decrease the associated cost is the Block Nested Loop Join (BNLJ). The improvement is that one table is only scanned once for each group of tuples of the other table. Particularly, suppose R is the inner table and S is the outer table. The difference between a nested loop join and a blocking one is that instead of every tuple in the inner table being compared to each tuple in other table, table S is loaded into buffer memory and then table R is scanned and the tuples in it are compared to the block of memory for a match to be found. Hence, this reduces the number of scans, and consequently, I/O cost. The memory is refilled by the remaining tuples in the outer table as many times as necessary and this process is repeated. The above procedure is summarized into two steps as below.

1) Divide table S into blocks for buffer memory.

2) Each tuple in table R is compared to the tuple in the block of the memory for a match to be found.

However, the cost is still a significant issue. As a result, in this project, a parallel approach for BNLJ is introduced. Specifically, we parallelize both tables such that the blocks of the outer table is distributed among memory of different cores and the chunks of rows in the inner table are handled by the threads in different cores. The proposed parallelization makes the steps 1 and 2 more efficient. Instead of comparing the entries in the block of memory one at a time, we can run multiple threads as well as considering different blocks simultaneously on the Phi multi-core processor.

# Experimental Results

In this section, we present our numerical results using our proposed approach. For comparison, we have compared our parallel BNLJ with parallel NLJ to show the impact of blocking on the running time. As we can see, in most cases, the blocking method slightly increase the timing performance. Also, the performance is improved when the number of threads increased from 10 to 20 which shows the need for a parallel framework. This shows that in the blocking approach we can benefit from cache availability by putting some parts of data input in the cache.