Amery Chang
ac2925
PFP COMS 4995.002 Fall 2022

Parallel Joins

(Please consider this complementary to the slides- the code walkthrough and description of the algorithms is much better represented on slides, so I avoided describing those things here since it would be a messy word salad)

Abstract

Database join operations tend to be among the most computationally expensive operations that databases perform. Joins can be implemented in Haskell to take advantage of Haskell's parallelism. Two commonly implemented join algorithms for database relations are the *nested loop join* and the *sorted-merge* (or just *merge*) join. My project will implement these algorithms in parallel in order to boost their performance compared to performing performing the joins in serial.

Background

After a SQL query is submitted, it is translated from the high-level SQL dialect into lower level query plan. Query operators, implemented in a lower-level programming language such as C, are based on Ted Codd's relational algebra, a small calculi that defines operations between relations (or less formally, tables). The relational algebra was seminal because it allows users to reason about table operations without knowing about technical implementation details- case in point, a user could ask the database to join two tables without knowing whether the database system will decide to use nested loop join or merge join. It's for this reason that database operators are able to reason about databases on a *logic* level (speaking the language of tables, data models, data manipulation and definition language) without dealing with the *physical* level (e.g. data structures, implementation details of algorithms, memory management).

Implementing a full on query execution engine and optimizer is out of scope for this project, as my goal is to focus on the join algorithms. However, it's interesting to note that even by implementing some basic functionality, I'm not too far away from applying some basic query optimization rules. For example, an optimization technique called *predicate pushdown*, where a filter (or predicate) is applied preemptively in order to process fewer total rows early on, could be implemented by applying my filter function. Additionally, I've built in a basic notion of cardinality on tables (i.e. a table with fewer rows is "less than" a table with more rows), which is integral for optimizing multiple joins.

Implementation

Joining rows can be made an embarrassingly parallel problem by partitioning the rows on the join key. By systematically creating partitions on some value of the join key, we ensure that the possible candidate rows to join with a given row must be within a particular partition.

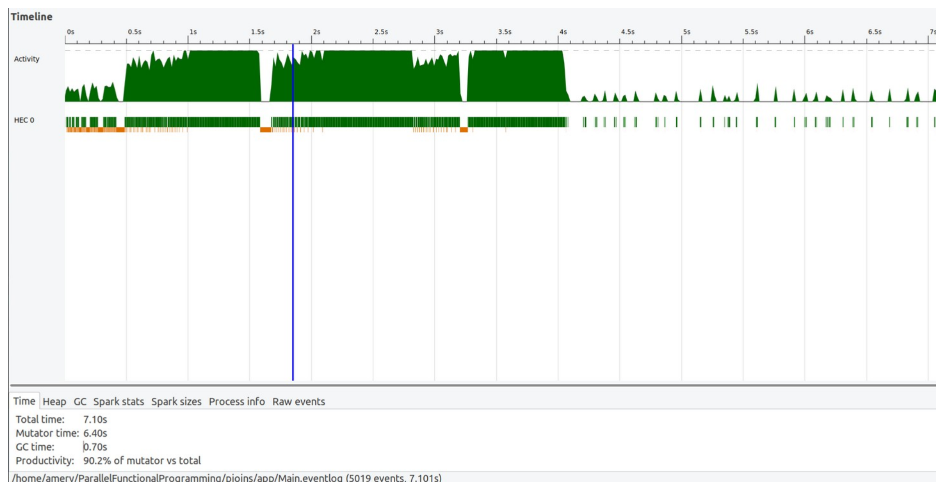
Given the above, an intuitive strategy for implementing parallelism on both the nested loop join and the merge join is as follows:

1. Take two tables to be joined, and partition it into *n* parts based on the join column
    - For example, if I'm joining on some String identifier, I can partition on the first character of that identifier
2. Perform the join algorithm on each partition. We should match each partition of the first table with its corresponding partition in the second table.
3. Once those partitions are joined, union all of the joined rows together into one table object.

With the above in mind, I chose to focus on the *dataflow* style of Haskell parallelism that implements the Par monad. This is rather different from the usage of the *Eval* monad and Strategies, which make use of Haskell's lazy evaluation. We use the Par monad's *spawnP* to create a child process for the join of each partition. An interesting implementation detail is that I did not need to modify the code for my join algorithms to make it run in parallel- the same code that operates on a table can also operate on a partition.

Results

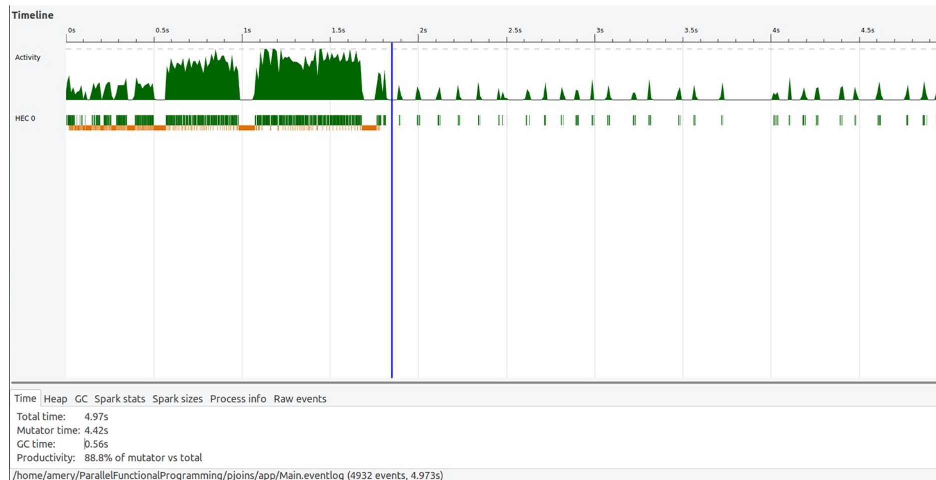
The parallel join implementations were tested by reading and joining two tables of financial records on a key.



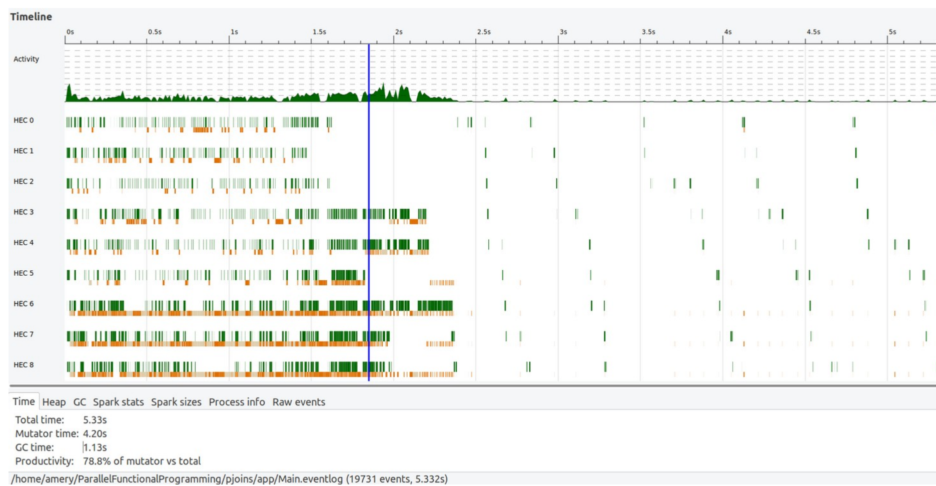
Nested Loop Join, no parallelism

Nested Loop Join, 9 parallel partitions

Korth, H. F., & Silberschatz, A. (1991). Database system concepts. McGraw-Hill.



Sort Merge Join, no parallelism



Sort Merge Join, 9 parallel partitions

References

1. Korth, H. F., & Silberschatz, A. (1991). *Database system concepts*. McGraw-Hill.
2. Marlow, S. (2013). *Parallel and concurrent programming in Haskell*. O'Reilly & Associates.