Data Processing on Modern Hardware
Assignment 6 – Task-Level Parallelism

Handout: **17th June 2020**
Due: **24th June 2020 by 9am**

Introduction

In Assignment 3 you have learned about different partitioning approaches for the input relations and combined them with a hash join to build a radix join, which is cache- and TLB-conscious. In this assignment we will further accelerate the radix join with multi-pass partitioning from Assignment 3 exploiting available parallelism on modern CPUs.

In the first part of this exercise we will focus on task-level parallelism (TLP) to take advantage of multiple cores (and, if available, multiple hardware threads) in your machine. In the second part of this exercise you will extend your TLP-implementation by NUMA-awareness, which we discussed in this weeks lecture. We provide you the implementation of the radix join from assignment 3 as foundation[1].

Part 1 - Implementation of a Parallel Radix Join

Your task is to speed up the radix join with multi-pass partitioning from Assignment 3 using task-level parallelism. As guideline, you can simply follow the basic steps that we suggest here.

Second-Level Partitioning

You should partition the relations in at least two passes. After the first pass, thread synchronization is a lot easier since the threads can operate on disjoint memory regions (partitions) in parallel. Therefore, we recommend that you start with the second-level partitioning first. To reduce the impact of load imbalance apply TLP. For each partition, create a separate task and store it in the queue. When a thread is done processing some partition, it fetches the next partition to work on from the task queue. You only need to synchronize the access to the task queue among the various threads.

First-Level Partitioning

The first partitioning phase of the radix join can be implemented in three steps. We will again use a task queue to distribute work among the various threads. At the end of each of the three steps an explicit barriers is required to synchronize the threads.

You can find a visualization of the following steps for partitioning with software-managed-buffers in lecture 8, slide 14 and a detailed description in chapter 6.1 of this paper[2]. Adapt it for multi-pass partitioning.

[1] https://gitlab.db.in.tum.de/dpmh-ss20/hw6
[2] https://dl.acm.org/doi/pdf/10.1145/2882903.2882917

Step 1 - Local Prefix Sums

Equally divide the input tuples amongst tasks (τ). Set the number of tasks ($\#\tau$) to $4 \times \#$threads. In practice, this is a good heuristic. For each task τ_i compute the local histogram $Hist_i$, i.e., count the local tuples in every bucket.

Step 2 - Merged Prefix Sums

The next step is to compute the *prefix sum* for every task in a parallel fashion based on the previously computed histograms. Consider a bucket with index j. For each task you need to compute the start address within the bucket corresponding to j. The total (global) number of tuples mapping to the j^{th} index is $\sum_{i=1}^{\#\tau} Hist_i[j]$. Thus, the start address of index j for some task i can be computed by adding up the histogram values of all indices less than j for all tasks plus the histogram values with index j for all tasks chronologically before the i^{th} task.

Step 3 - Partitioning

Finally, each task iterates over its share of tuples a second time and uses its local prefix sum, derived in step 2, to scatter the tuples to their final locations.

Part 2 - Implementation of a NUMA-Aware Parallel Radix Join

In the last lecture, we discussed NUMA architecture and its effects. Adapt your implementation of the parallel radix join from *Part 1* for a NUMA-architecture and make it NUMA-aware. Again, have a look at lecture 8, slide 14 and at chapter 6.1 of this paper[3].

Part 3 - Analysis and Evaluation

Eventually, evaluate your implementations for the following aspects:

- Vary the #threads and measure the speed-up to investigate the scalability of your code.
- Compare the performance of the non-NUMA-aware implementation vs. the NUMA-aware implementation.
- Vary the #tasks from fine-grained (small) to large tasks. Do the results match your expectations?

Submission Guidelines

This homework has a duration of one week. Fork the repository, commit your changes in the git, and invite us (@*dpmh*) to hand in your homework.

The programming language of this homework is C++. We provide you a code skeleton, add functions needed for the implementation. For performance measuring of the experiments, you can either use the provided `perfEvent.hpp` or use the tools you applied in the previous assignments.

[3] https://dl.acm.org/doi/pdf/10.1145/2882903.2882917