5 Parallel Partitioned Hash Join

In Assignment 3 you learned all about implementing in-memory *hash join* algorithms that are cache- and TLB-conscious. In this assignment we will further accelerate the *(multipass) partitioned hash join* from Assignment 3 exploiting available parallelism on modern CPUs. In the first part of this exercise you will focus on *thread-level parallelism* (TLP) to take advantage of multiple cores (and, if available, multiple hardware threads) in your machine. The second part concentrates on *data-level parallelism* (DLP), where you can use your knowledge from Assignment 4 about SIMD and SSE/AVX to further accelerate the join operator. You can either parallelize your own code from Assignment 3 or a *serial version* of the *Radix-Cluster Hash Join*, which can be downloaded here:

http://www.systems.ethz.ch/sites/default/files/file/dpmh_Fall2012/src-handout-05_tar.bz2.

This assignment is partially based on the paper [1] by C. Kim et al., in which the authors claim that their implementation can join more than 100 million tuples per second. Thus, your goal should be to come as close to this number as possible.

5.1 Task

Your task is to speed up the partitioned hash join algorithm from Assignment 3 as much as possible using hardware parallelism. How fast can you join two relations of 100 million tuples each? You are free to do whatever optimization you can think of and/or you can simply follow the basic steps that we suggest here.

5.2 Second-Level Partitioning (TLP)

With 100 million tuples, 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 the *Task Queuing model* [2]. 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.

5.3 First-Level Partitioning (TLP)

The first partitioning phase of the partitioned hash join algorithm can be implemented in three steps. We will again use a *task queue* [2] 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.

5.3.1 Step 1

Equally divide the input tuples amongst τ' tasks. Set $\tau' = 4\tau$, where τ is the number of 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.

5.3.2 Step 2

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 number of tuples mapping to the j^{th} index is $\sum Hist_i[j]$. 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.

5.3.3 Step 3

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

5.4 Nested-Loops Join (DLP)

With small buckets *nested-loops join* can be very fast (see Assignment 3). Thus, we use nested-loops in this exercise to join the partitions, which we make very small. Using SSE/AVX intrinsics, you should be able to speed up nested-loops such that you can join larger partitions efficiently. Larger partitions means less partitioning overhead, which should result in an overall speedup.
In [3], three ways to parallelize nested-loops using SIMD are presented: *duplicate-inner*, *duplicate-outer*, and *rotate-inner*. In *duplicate-inner* and *duplicate-outer* one element of one of the two relations is replicated to make a SIMD unit. This unit is then compared against a SIMD unit of tuples from the other relation. In *rotate-inner*, by contrast, no tuples are replicated but one SIMD unit is rotated n (*e.g.*, 4) times and compared against the other SIMD unit after each rotation. Use any of these methods to accelerate the nested-loops algorithm in the code.

5.5 Optional: find further data-level parallelism (DLP)

Other then the nested-loops join, are there more opportunities in your program to exploit data level parallelism (DLP)? For instance, can you speedup the computation of the histograms using SIMD? What about the prefix sum calculation? Anything that gets you closer to joining 100 million tuples per second is a valid approach!

References

[1] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *PVLDB*, 2(2):1378–1389, 2009.

[2] Eric Mohr, David A. Kranz, and Robert H. Halstead Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 2(3):264–280, 1991.

[3] Jingren Zhou and Kenneth A. Ross. Implementing database operations using simd instructions. In *SIGMOD Conference*, pages 145–156, 2002.