Research plan for Nina Wang

Jelle Hellings

Department of Computing and Software
Information Technology Building (ITB), room 124
1280 Main Street West, Hamilton, ON L8S 4L7, Canada

End goal A parallel temporal join algorithm.

A temporal dataset is a set of events of the form (*begin*, *end*) where *begin* is the timepoint at which the event starts and *end* is the timepoint at which the event ends.

Given two temporal datasets M and N, the temporal join $M \bowtie N$ is defined by

$$M \bowtie N = \{((b_1, e_1), (b_2, e_2)) \mid (b_1, e_1) \text{ overlaps with } (b_2, e_2)\}.$$

A join algorithm is parallel if we can speed up the algorithm by providing it access to more CPU cores to operate on.

Plan We aim at an algorithm that combines the ideas of a parallel merge sort and of SkipJoin. As such, our design is distinct from the approach in *A forward scan based plane sweep algorithm for parallel interval joins*.

A parallel merge sort works as follows:

Algorithm $\underline{\textsc{PMergeSort}}(L, n)$ **:**

Pre: L is an *array*, n is the number of threads we can use.

Divide L into n roughly-equally-sized pieces $L_1, \dots, L_n$.
Use a high-performance sort to sort each of $L_1, \dots, L_n$ in parallel.
Let $X = [L_1, \dots, L_n]$.
while $|X| > 1$ **do**
    Let $Y = []$.
    while $|X| \geq 2$ **do**
        Choose and remove the first two lists M_1 and M_2 in X.
        Merge M_1 and M_2 together with an n-parallel merge algorithm and add the result to Y.
    end while
    Add $X[0]$ to Y if $|X| = 1$.
    $X := Y$.
end while
return $X[0]$.

In parallel merge sort, we need to be able to merge in a parallel manner. For this, we need an n-parallel merge algorithm. Such an algorithm works as follows (very high level):

Algorithm $\underline{\textsc{PMerge}}(M_1, M_2, n)$ **:**

Pre: M_1, M_2 are *sorted array*, n is the number of threads we can use.

Let R be an array of size $z = |M_1| + |M_2|$.
Find the values $v_0, v_1, \dots, v_{n-1}, v_n \in (M_1 \cup M_2)$ such that v_i is $i \cdot \frac{z}{n}$-th smallest value in $M_1 \cup M_2$.
Divide M_1 and M_2 into pieces $M_{1,1}, \dots, M_{1,n}$ and $M_{2,1}, \dots, M_{2,n}$ such that the values in piece $M_{j,i}$ is between v_{i-1} and v_i in M_j.
Use a high-performance merge to merge each $M_{1,j}$ and $M_{2,j}$ into $R[i \cdot \frac{z}{n} \dots (i+1) \cdot \frac{z}{n} - 1]$ in parallel.

The important step in the avove is finding the values $v_1, \ldots, v_{n-1}$, which we can do with a clever binary-search-like algorithm.

In parallel merge join, we use similar ideas: we compute parts of the join in parallel. The main difference with parallel merge is that in merge join *we do not know the size of each output*.

In a parallel temporal join, we need a bit more than in a parallel merge join: one cannot simply split two lists of events M and N into lists M_1, M_2, N_1, N_2 such that $M \bowtie N$ is equal to $M_1 \bowtie N_1 \cup M_2 \bowtie N_2$: events in that begin early (and, hence, are in M_1 and N_1) might have a very long duration and end after events in M_2 and N_2 end. To find such events, we will use the approach used by SkipJoin: we use an interval tree index to find those events.