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 <u>**PMergeSort**</u>(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 <u>**PMerge**</u>(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 above is finding the values $v_1, \ldots, v_{n-1}$, which we can do with a clever binary-search-like algorithm.

Assume we want to merge two lists M_1 and M_2 into a target M of size $|M_1| + |M_2|$. We want to do so with two threads that each merge $\frac{|M_1|+|M_2|}{2}$ values. We do so by findign the median m of M_1 and M_2. Next, the first thread will merge all values *smaller than* the median and the second thread will merge all values *larger than* the median. Next, I will detail how to get the median of two lists.

Analysis. First, we assume that the median is in M_1 and not in M_2. We also assume that $(|M_1| + |M_2|)$ is odd and all values are distinct. Hence, there is exactly one median value (I leave it as an exercise to find the median without these two restrictions).

A value $m \in M_1$ is the median if $E = \lfloor(|M_1| + |M_1|)/2\rfloor$ values in $M_1 \cup M_2$ are smaller than m and E values are larger than m. Assume the median is at position p in M_1, $0 \leq p < |M_1|$, and that we are currently inspecting a position i, $0 \leq i < |M_1|$. If we *inspect* position i, we already know:

$$P : i \text{ values in } M_1 \text{ are smaller than } M_1[i], \text{ as } M_1 \text{ is sorted.}$$

Hence, if i is the position of the median, then $E - i$ values in M_2 need to be smaller than $M_1[i]$ and all other values in M_2 need to be larger than $M_1[i]$. We can check whether these two conditions are true by comparing $M_1[i]$ with the values at $M_2[E - i - 1]$ and $M_2[E - i]$.

Note that $E - i - 1$ and $E - i$ are not necessary valid positions in M_2 (the positions $0, \ldots, |M_2| - 1$). To simplify notation here, we assume that for all positions $x < 0$ in M_2, we have $M_2[x] = -\infty$ (smaller than any value) and that for all positions $x \geq |M_2|$, we have $M_2[x] = \infty$ (larger than any value). We have:

1. $M_1[i] < M_2[E - i - 1]$: as the list M_2 is sorted, less than $E - i$ values in M_2 are smaller than $M_1[i]$. Hence, $M_1[i]$ is too small to be the median. If the median is in M_1, it has to be at a position larger than i.

2. $M_2[E - i] < M_1[i]$: as the list M_2 is sorted, more than $E - i$ values in M_2 are smaller than $M_1[i]$. Hence, $M_1[i]$ is too large to be the median. If the median is in M_1, it has to be at a position smaller than i.

3. $M_2[E - i - 1] < M_1[i] < M_2[E - i]$: exactly $E - i$ values in M_2 are smaller than $M_1[i]$. Hence, $M_1[i]$ is the median.

Hence, using two comparisons of $M_1[i]$ (with $M_2[E - i - 1]$ and $M_2[E - i]$), we can determine whether we found the median or whether we need to search left of i or right of i. Hence, we can use a variant of binary search to find the position in M_1 of the median:

Algorithm <u>FindMedianIfInM_1</u>(M_1, M_2) **:**

Pre: M_1 and M_2 are ordered *arrays*, the median is in M_1.

begin, *end* := $0, |L|$.
$E := \lfloor(|M_1| + |M_1|)/2\rfloor$.
while *begin* $\neq$ *end* **do**
    mid := $\lfloor(begin + end)/2\rfloor$.
    if $M_1[mid] < M_2[E - mid - 1]$ **then**
        begin := $mid + 1$.
    else if $M_2[E - mid] < M_1[mid]$ **then**
        end := *mid*.
    else if $M_2[E - mid - 1] < M_1[mid] < M_2[E - mid]$ **then**
        return $M_1[mid]$.
    end if
end while
/* We should never reach here, as the median is assumed to be in M_1 */.

Finally, we have to deal with the case in which the median is in M_2. In that case, no value for *mid* will ever satisfy the conditions of Line 9. The search will always reduce the difference between *begin* and *end*. Hence, eventually we end up with *begin* = *end* and reach Line 13. At that point, we *must* have the median in M_2. Hence, we can swap the role of M_1 and M_2 to get the correct outcome:

Solution.

Algorithm <u>FindMedian</u>(M_1, M_2) **:**

Pre: M_1 and M_2 are ordered *arrays*, the median is in M_1.

1: $begin, end := 0, |L|$.
2: $E := \lfloor(|M_1| + |M_1|)/2\rfloor$.
3: **while** $begin \neq end$ **do**
4:     $mid := \lfloor(begin + end)/2\rfloor$.
5:     **if** $M_1[mid] < M_2[E - mid - 1]$ **then**
6:         $begin := mid + 1$.
7:     **else if** $M_2[E - mid] < M_1[mid]$ **then**
8:         $end := mid$.
9:     **else if** $M_2[E - mid - 1] < M_1[mid] < M_2[E - mid]$ **then**
10:         **return** $M_1[mid]$.
11:     **end if**
12: **end while**
13: **return** FindMedian(M_2, M_1).

Note: in this algorithm, we use the assumption that values in a list L before position 0 have the value $-\infty$ and that values in a list L after position $|L| - 1$ have the value ∞.

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.

Plan for attack: lets work out the full design of the following algorithms:

1. Lets implement a two-thread merge sort as a practice algorithm (alongside a single-threaded algorithm).
2. Then, lets implement a two-thread merge join as a second practice step (alongside a single-threaded algorithm).
3. Finally, let see what we need to go from that merge join to a temporal join and provide the methods to do so.

1 Parallel Natural Joins

Assume we have two tables $T_1(A, B)$ and $T_2(A, C)$ and we want to compute $T_1 \bowtie T_2$. Hence, we want to compute $T_1 \bowtie T_2 = \{(a, b, c) \mid (a, b) \in T_1 \land (a, c) \in T_2\}$. Now let

$$T_1 = T_{1,1} \cup \ldots T_{1,n} \qquad\qquad T_2 = T_{2,1} \cup \ldots T_{1,m}$$

As the natural join distributes over union, we have

$$T_1 \bowtie T_2 = (T_{1,1} \bowtie T_{2,1}) \cup \ldots \cup (T_{1,1} \bowtie T_{2,m}) \cup \ldots \cup (T_{1,n} \bowtie T_{2,1}) \cup \ldots \cup (T_{1,n} \bowtie T_{2,m}).$$

Clearly, we can compute the natural join T_1 and T_2 in parallel using $n \cdot m$ threads by computing the above $n \cdot m$ individual joins in parallel.

The above approach is rather brute force: it does not take into account any structure in the datasets T_1 and T_2. Now consider both T_1 and T_2 are ordered on increasing value of attribute A and let

$$T_1 = T_{1,1} \cup T_{1,2} \qquad\qquad T_2 = T_{2,1} \cup T_{2,2}$$

such that $T_{1,1}$ and $T_{2,1}$ contain exactly those values in T_1 and T_2 with a value for attribute A that is smaller-or-equal to some constant c. By the above, we have

$$T_1 \bowtie T_2 = (T_{1,1} \bowtie T_{2,1}) \cup \ldots (T_{1,1} \bowtie T_{2,2}) \cup (T_{2,1} \bowtie T_{2,1}) \cup \ldots \cup (T_{2,1} \bowtie T_{2,2}).$$

Due to the structure of T_1 and T_2, we know $(T_{1,1} \bowtie T_{2,2}) = \emptyset$ and $(T_{2,1} \bowtie T_{2,1}) = \emptyset$. Hence, we can simplify the above to

$$T_1 \bowtie T_2 = (T_{1,1} \bowtie T_{2,1}) \cup (T_{2,1} \bowtie T_{2,2}),$$

which allows us to compute $T_1 \bowtie T_2$ efficiently using two threads. We note that the above approach is best if both join tasks are roughly an equal amount of work. This is hard to assure, however: some values of attribute A might yield many more rows in the output than others.

There are several strategies to deal with the above:

1. Assuming we have n threads available: split the task $T_1 \bowtie T_2$ up into much-more than n tasks and let the n threads pick tasks from a queue of still-to-compute joins. Now if a thread t has a heavy task, then the other threads can do multiple tasks while t is taking care of the heavy task. (Note: this does not deal with the case in which the heavy task of t is the last task in the queue, but as we have many tasks, the impact of a heavy task will be lower than if we only made exactly n tasks.)

2. We assume that the join result is equally distributed over all values of A. In this case, we simply have to assure that $|T_{1,1} \cup T_{2,1}| \approx |T_{1,2} \cup T_{2,2}|$ (hence, both join tasks process roughly the same amount of data). To do so, we simply split T_1 and T_2 based on the median value for attribute A in $T_1 \cup T_2$.

2 Temporal Natural Joins

A time-interval is a pair (s, e) in which s is a start-time and e is an end-time. Time t is in (s, e) if $s \leq t \leq e$. The time-intervals $i_1 = (s_1, e_1)$ and $i_2 = (s_2, e_2)$ overlap if there exists a time t with $t \in i_1$ and $t \in i_2$. We write overlaps(i_1, i_2) to denote the predicate that holds if and only if i_1 and i_2 overlap.

Now assume we have two tables $T_1(A, B)$ and $T_2(A, C)$ such that attribute A holds time-intervals. The temporal join $T_1 \bowtie_{\odot} T_2$ is defined by $T_1 \bowtie_{\odot} T_2 = \{(a_1, b, a_2, c) \mid (a, b) \in T_1 \land (a_2, c) \in T_2 \land \text{overlaps}(a_1, a_2)\}$. Let

$$T_1 = T_{1,1} \cup T_{1,2} \qquad\qquad T_2 = T_{2,1} \cup T_{2,2}$$

such that $T_{1,1}$ and $T_{2,1}$ contain exactly those values in T_1 and T_2 with a time-interval for attribute A that starts before some time t.

As the temporal join distributes over union, we have $T_1 \bowtie_{\odot} T_2 = (T_{1,1} \bowtie_{\odot} T_{2,1}) \cup (T_{1,1} \bowtie_{\odot} T_{2,2}) \cup (T_{1,2} \bowtie_{\odot} T_{2,1}) \cup (T_{2,1} \bowtie_{\odot} T_{2,2})$. In this case, we do not necessary have $(T_{1,1} \bowtie_{\odot} T_{2,2}) = \emptyset$ or $(T_{1,2} \bowtie_{\odot} T_{2,1}) = \emptyset$, however. Assume $t = 12$ and consider a row $(a, b) \in T_{1,1}$ with $a = (7, 15)$. Even though time-interval a starts before t (and, hence, (a, b) must be in $T_{1,1}$), we still can have $\{(a, b)\} \bowtie_{\odot} T_{2,2} = \emptyset$.

Due to the above observation, we cannot use the approach of Section 1 to compute $T_1 \bowtie_{\odot} T_2$ in few parallel tasks: if we break up the computation of $T_1 \bowtie_{\odot} T_2$ into $(T_{1,1} \bowtie_{\odot} T_{2,1}) \cup (T_{1,1} \bowtie_{\odot} T_{2,2}) \cup (T_{1,2} \bowtie_{\odot} T_{2,1}) \cup (T_{2,1} \bowtie_{\odot} T_{2,2})$, then we need a way to determine $(T_{1,1} \bowtie_{\odot} T_{2,2})$ and $(T_{1,2} \bowtie_{\odot} T_{2,1})$, this preferably without computing these from scratch.

Consider $(T_{1,1} \bowtie_{\oplus} T_{2,2})$. By construction, all time-intervals in $T_{1,1}$ start before t and all time-intervals in $T_{2,2}$ start at-or-after t. Hence, $(a_1, b, a_2, c) \in (T_{1,1} \bowtie_{\oplus} T_{2,2})$ only if $a_1 = (s_1, e_1)$ ends after $a_2 = (s_2, e_2)$ starts: we must have $s_1 < t \leq s_2 \leq e_1$.

Let $T|_t = \{(s, e) \in T \mid s \leq t \leq e\}$ be all time-intervals in T that hold time t. By the above, we must have $(T_{1,1} \bowtie_{\oplus} T_{2,2}) = T_{1,1}|_t \bowtie_{\oplus} T_{2,2}$ and, likewise, $(T_{1,2} \bowtie_{\oplus} T_{2,1}) = T_{1,2} \bowtie_{\oplus} T_{2,1}|_t$.

Given an interval-based dataset T ordered lexicographically on start and end times (hence, time-intervals are ordered on start times and time-intervals with equal start-time are ordered on end times), we can build an index structure that allows us to easily compute stab queries of the form $T|_t$. We refer to https://doi.org/10.4230/LIPIcs.TIME.2020.18 for details on how a stab forest operates and we refer to https://www.jhellings.nl/projects/skipjoin/ for implementation details.

We note that if we assume that T never changes, then we can even easier build a stab-forest-like index of T that allows us to answer stab queries of the form $T|_t$: we simply build a tree on start-times such that each node labeled with start-time s points to a left child representing all time-intervals that start at-or-before s, a right child representing all time-intervals that start after s, and an *left list* index structure pointing to all time-intervals represented by the tree rooted at the left child that end after s such that the time-intervals in the *left list* are ordered on end times.