# Assignment 3

**Due** Thursday by 10pm **Points** 10

Assignment 3 - Shared Memory OpenMP
**Due**: Thursday, March 11, at 10 p.m.

**Points**: 10

**Deadline to find your group partner and create your group on MarkUs**: **Monday, March 1, at 10:00 pm**. Your group partners for assignment 3 and lab 4 remain the same. If you do not have a partner in MarkUs by this deadline, you will not be able to submit this assignment!

## Scinet

We will use Scinet for this assignment. You should have already received an email with your Scinet login. In that email you see a link to *The short intro to Scine*t document (the document is also uploaded in Quercus). You have to read that document before starting this lab. While you can use your own machine or a lab machine to debug the code, your final code has to be performance engineered and executed on Scinet. We will do all grading on Scinet. For each part, we do provide scripts to help with running your code on Scinet.

## Overview

For this assignment, you will employ shared memory techniques to parallelize database join operations. Your task is to implement the join operation in parallel in OpenMP.

You should work with a partner on this assignment. Please log into MarkUs as soon as possible to find your repository and invite your partner, and make sure that you can commit and push to your repo. For all assignments and labs you will be given your repo URL on MarkUs. Make sure to use this repository (which should already be created for you), otherwise MarkUs won't know about it and we won't be able to see your work.

The starter code is available on Scinet under /home/t/teachcsc367/CSC367Starter/assignments/assignment3/starter.tgz. This tarball includes the starter code, under the starter_code directory, as well as the datasets which you will use to run your experiments, under the data directory. First, clone your repository to your $SCRATCH directory. Copy the starter code (i.e. the contents of starter_code) into your repository and make sure you can do your first commit (please only commit code - do not commit any of the datasets). You will then have to place the data folder in the same directory (i.e. your $SCRATCH directory) as your repository, such that the path to it from your repository is "../data". Please make sure to read carefully over the code, including the licenses and the instructions from the comments.

## Database Joins in a nutshell

Joins are some of the most common operators in database queries, and some of the most time consuming operations in large scale data processing. The join operation is denoted by a ⋈ symbol and combines two relations, R and S, to form a new relation T. Considering the attributes (columns) of R as r1, r2, ..., rn, and the attributes of S as s1, s2, ..., sm, the join operator creates a new relation T, containing all the n+m attributes r1, r2, ..., rn, s1, s2, ..., sm. In general such queries involve a projection which retains into T only those attributes of interest from the joined relations R and S. Additionally, it is common for one or more join predicates has to satisfied by tuples in R and S, before they can be added to the new relation T.

One such type of join is known as an equi-join, where tuples in R and S have to match the values of one or more attributes. For example, consider two relations Student(id, name, year) and Staff(id, name, salary). An example join query between these two relations could be formulated as:

```
SELECT R.name, S.year FROM Student R, Staff S
WHERE R.id = S.id;
```

This query finds all students who are also staff members of the university, and outputs their names and year of study. Joins can be implemented in several ways. Some of the more common approaches are called nested-loop join, sort-merge join, and hash-join.

## 1. Nested-loop join

The nested-loop approach is the simplest join technique. One of the relations (let's say R) is designated to be the outer relation, while the other one (in our case, S), is considered the inner relation. For each tuple in the outer relation R, we compare all the tuples from the inner relation S. All the tuple pairs (r,s) that satisfy the join predicate P are added to the output relation T. Note that the relations can be joined on multiple predicates as well. The nested loop join approach can be formulated in pseudocode as:

```
for each tuple r in R do
  for each tuple s in S do
    if r and s satisfy P then
```

```
            add (r,s) to T
         endif
      endfor
   endfor
```

While the nested loop technique is simple to implement and to reason about, it is a somewhat brute-force approach which performs an exhaustive search over the cartesian product between R and S. **As a side-note, some optimizations in the database domain include using search trees (such as B-trees) to find matches faster than sequentially scanning through a relation.** The query optimizer is a component which decides the query plan, in terms of which join approach to use, what indexes (if any) need to be used, etc. This decision is quite complex and typically depends on a set of statistics about the relations (e.g., relation sizes, distribution of attribute values, etc.), as well as system resources (e.g., memory space).

## 2. Sort-merge join

The sort-merge approach is a join technique that avoids the exhaustive search of a nested loop join, provided that the relations are sorted on the join attribute(s). If both relations are sorted in the order of the join attribute, tuples that satisfy the join predicate(s) are much faster to find. The tuples that satisfy the join predicate(s) are merged to form the result T. The sort-merge join approach can be formulated in pseudocode as:

```
   sort R on attribute x
   sort S on attribute y

   r = first tuple in R
   s = first tuple in S

   while (r did not reach the end of R) and (s did not reach the end of S)
      if r.x > s.y then
         advance s to next tuple in S
      else if r.x < s.y then
         advance r to next tuple in R
      else
         /* found a match for equi-join */
         add (r,s) to T

         r' = next tuple in R after r
         while r' not at end of R and r'.x == s.y
            add (r',s) to T
            r' = next tuple in R after r'
        endwhile

        s' = next tuple in S after s
        while s' not at end of S and r.x == s'.y
            add (r,s') to T
            s' = next tuple in S after s'
        endwhile

        r = r'
        s = s'
    endif
endwhile
```

The main advantage of the sort-merge approach is that the search complexity drops from n x m operations (in the case of a nested-loop join) to n + m operations (where n = |R|, and m = |S|, the cardinalities of the two relations). As a side note, if the relations are known to be joined frequently, the database administrator will typically keep the relations sorted, in order to avoid the cost of the sorting step. This algorithm works really well, especially in the absence of any indices on the joined relations.

## 3. Hash join

The hash join approach is a join technique that does not rely on the relations being sorted beforehand. Instead, we use a hash function on the join attributes, to enable a faster lookup for matches. For example, a simple hash join approach would be to hash relation R on attribute x, and then start reading tuples from relation S, one at a time. By applying the same hash function to each tuple s on the join attribute y, we determine which tuple in R (if any) are a match. This simple hash join approach can be formulated in pseudocode as:

```
for each tuple r in R do
    hash r into hashtable HR
endfor

/* Lookup in S */
for each tuple s in S do
    lookup hash of s in HR
        if bucket not empty then
            for each tuple r in bucket
                if r.x == s.y then
                    add (r,s) into T
                endif
            endfor
        endif
    endfor
```

Note: the algorithm can be simplified if the the join attribute is a (unique) primary key in one of the relations (say R). In such case, you can make a hash table that maps primary key values to the corresponding tuple in R, and then simply iterate through S, looking up the join attribute value in the hash table. Feel free to use your (single-thread) hash table implementation from Lab3, it should be sufficient.
One of the advantages of the hash join approach is that if the hash table for one of the relations fits in memory, the join only has to go over the other relation to fetch tuples from disk. In the algorithm above, we usually pick R to be the smaller of the relations.
This approach can be modified to hash both relations, which basically partitions both of the relations into buckets R1, R2, ..., Rk, and S1, S2, ..., Sk. Since the hash function is the same, we have the same number of buckets, and each bucket pair Ri and Si will contain the same range of values. The join now becomes a matter of doing a bunch of much smaller (partial) joins between each of the buckets that correspond to the same range (R1 ⋈ S1, R2 ⋈ S2, etc.). For each of these bucket pairs, we can perform the partial joins using either a nested-loop, a sort-merge, or even a finer-grained hash function approach.

This may be obvious, but it's worth noting that the choice of hash function is important for the performance of the hash join. A good hash function will distribute the values uniformly, whereas a poor choice of hash function could end up creating very imbalanced buckets, with some being empty and others holding a lot of records.

## 4. Parallel Join

The performance of a join operation can be a limiting factor if the relations involved contain a large amount of data. Luckily, the join operation has an inherent degree of parallelism which can be exploited to speed up its execution. To this end, two approaches that can be taken are called fragment-and-replicate and symmetric partitioning (description and additional references below).

**Fragment-and-replicate**
The fragment-and-replicate approach, as the name suggests, partitions one of the relations R over all the processing elements (cores on the same machine), and replicates the other relation S in its entirety over all the processing elements. The join is performed by joining each fragment of R with the fully replicated relation S. This method works really well when R is very large and S is relatively small. Naturally, replicating the larger relation and partitioning the smaller one is less efficient. Consequently, this method requires that one of the relations be small enough to keep the replication cost reasonable. Otherwise, distributing a very large relation over the interconnect between the processing elements can be a major bottleneck.

Each of the partial joins between a fragment of R and the relation S can be performed using either a nested-loop, a sort-merge, or a hash-join approach.

**Symmetric partitioning**
Another approach, called symmetric partitioning, partitions both relations such that each fragment Ri only needs to be joined with one fragment Si. As a result of the symmetric partitioning, the partial joins between pairs of fragments Ri - Si, can be done independently from other fragment joins. Therefore, these partial joins can be processed in parallel, without any dependencies.

This approach can be applied to parallelize any of the sequential join techniques described above (nested-loop, sort-merge, hash join). For example, the hash join approach described above will guarantee that fragment pairs can be joined independently.

References: Query evaluation techniques for large databases **(http://dl.acm.org/citation.cfm?id=152611)**

## Getting started

In this assignment you will implement a join query with the following specifications. Consider that you are given two relations, stored in a file, with the following schema:

```
Student(sid int, name char[20], gpa float)
TA(cid int, sid int, course char[8])
```

The 'sid' field (in both relations) is the student ID, which is the primary key in the 'Student' relation. The 'cid' field is the TA contract ID (for a TA in a particular course), which is the primary key in the 'TA' relation.
You will assume that no indexes are available on the given relations.
The query you have to implement finds all the students who have a GPA above 3.0 and are also employed as a TA by the university, and returns the number of matches. (Note: the number of matches is the number of TA contracts, not the number of students.) In SQL, the query would be formulated as follows:

```
SELECT count(*) FROM Student s, TA t
WHERE s.sid = t.sid AND s.gpa > 3.0;
```

You will implement 2 versions of the program that evaluates this query: sequential and shared-memory parallel using OpenMP.

The starter.tgz file has two folders, data and starter_code.

The datasets you will use for your experiments are stored in the data folder.
The starter code, under the starter_code folder, includes (among other things):

- Definition of C structures representing the data records - data.h.
- Functions to load and store the data - data.c.
- Command line argument parser (options.c).
  NOTE: do not change the command line interface of the programs.
- Skeletons of different (sequential) join algorithms - join.c
- Sequential version of the program - join-seq.c.
- You should use it to make sure that your sequential algorithms are implemented correctly **before** you start working on the parallel versions. It will also give you an important comparison point for the performance analysis in your report.
- Skeletons of the parallel join implementations (join-omp.c), including all necessary initialization boilerplate.

IMPORTANT: you must keep the same output format as in the starter code (in each join-*.c program). You must not print anything else to stdout, otherwise you will fail the tests. You can print error messages (e.g. failed memory allocations) to stderr. You may also print some debugging output to stderr during development, but you must remove all debugging output in your final submission.

## Task 1 - Sequential Join

In the first part, you must implement the given query using the three sequential join algorithms discussed above (nested loop, sort-merge, and hash join).

## Task 2 - Shared memory parallel join

In this part, you will use OpenMP to implement the given query using both the parallel join approaches discussed above in section 4 (fragment-and-replicate and symmetric partitioning).

Note: You may assume that both relations are already sorted by the sid. You should take advantage of the fact that relations are sorted in order to efficiently compute symmetric partitions.

For the fragment-and-replicate, you must partition the data in a way that balances the load among all processing units. For the partial joins, you must support the nested loop approach as well the sort-merge and the hash join, and compare them in your report.

Hint: be careful about how exactly you choose the "smaller" partition (the one to be replicated). There are 2 basic options, and each one of them can be better than the other in particular circumstances (e.g. depending on which join algorithm is used).

For the symmetric partitioning technique, you must ensure that partitioning is done such that both relations can be joined independently. You might want to analyze some of the tradeoffs involved. Note: In practice, a query optimizer keeps histograms about the distribution of values for each attribute. You may not assume that this is available, so you should describe in your report how you experimented with partitioning techniques and what you noticed. Hint: remember the techniques for dealing with imbalanced workloads that you've learned in the course.

NOTE: You must look into the pragma clauses that we discussed in class, as well as the OpenMP specifications, to fully take advantage of the automated parallelization abilities that OpenMP provides. Grading will put a considerable emphasis on efficiency.

## Data collection and analysis

We provide you a few datasets for testing and measuring performance of your code. These are included in the starter code tarball. You should store the data in your $SCRATCH directory on SciNet (teach cluster). Try to keep only one data copy under your $SCRATCH directory to avoid wasting your disk quota by duplicating the dataset. You are not allowed to use your own custom datasets to report the performance in your report.

| Dataset | Number of students in the ‘Student’ relation | Number of TAs in the ‘TA’ relation |
|---|---|---|
| 0 | 1000 | 484 |
| 1 | 5000 | 2106 |
| 2 | 20000 | 74950 |
| 3 | 10000000 | 4998844 |
| | | |

| | | |
|---|---|---|
| 4 | 10000000 | 250000420 |
| 5 | 10000000 | 10201456 |

Table 1: The description of the provided dataset

Table 1 provides details about the datasets. Dataset 0 is a small dataset, which should be used for debugging your code. You only need to report the performance of the serial and parallel implementation of the nest-loop join on datasets 0-2.

Don't run nested-join implementations (for either sequential or parallel joins) on larger datasets 3-5, which takes more than 4 hours to finish.

As a simple sanity test, we give you the correct output for dataset 0 and dataset 1 below. You should carefully implement the joins such that your output for the larger datasets is correct.

| | |
|---|---|
| Dataset 0 | 265 |
| Dataset 1 | 2106 |

Once you have implemented all the parts of the assignment and verified the correctness of the output, you must measure the time taken in each of the previous parts, and collect all the necessary data points.

You must represent your results visually (graphs!) and analyze them in a short report (you must name this file: report.pdf). You should describe your implementation, plot and analyze your results, draw conclusions and report your findings in a written report. It is important to show insight into what you are measuring and explain the performance gaps you are seeing.

The expected graphs in your report are as follows:

1. Use graphs to show the running time of the different join methods for the sequential implementations for datasets 0 to 2.
2. For datasets 0-5, use graphs to compare the running time of the following pairs of parallel and partial join methods:

- fragment-and-replicate, hash join
- fragment-and-replicate, sort-merge join
- symmetric partitioning, hash join
- symmetric partitioning, sort-merge join
- Provide scalability plots that show how the performance of the parallel joins scale for 1, 2, 4, and 8 threads. Speedups in the scalability graphs should be measured by comparing the parallel performance of the method with the single-threaded implementation of the same method. For example, the speedup of fragment-and-replicate with hash join on 4 threads should be measured

against the performance of fragment-and-replicate with hash join on 1 thread. The experiments should be conducted for datasets 0-5. Provide analysis for these scalability graphs. Again, provide these graphs for the following pairs of parallel and partial join methods:

- fragment-and-replicate, hash join
- fragment-and-replicate, sort-merge join
- symmetric partitioning, hash join
- symmetric partitioning, sort-merge join

You should discuss how various considerations discussed in class play a factor into the results.

## Testing tips

Note1: For this assignment, you are required to use teach cluster (SciNet) to get accurate performance results.

Note2: After logging in to Scinet and cloning your code into your *$SCRATCH* folder, you can start working on your assignment. We strongly recommend to study the Scinet introduction document before doing this lab to ensure follow the Scinet policies along running your code. You should run the assignment on Scinet compute nodes. Two scripts named with *run-job-a3-seq.sh* and *run-job-a3-omp.sh* are provided in order for running the sequential and OpenMP parts on Scinet compute node. Note that these scripts expect the datasets to be in the ../data/ folder, so please place the data folder in the same folder as your repository.

For the OpenMP part, you should set options in job-a3-omp.sh to run your preferred variant of the join algorithm.

The test report will include the output (stdout and stderr) of your code, so please make sure to keep your output within reasonable means (if the report file is larger than a few MB, it may not be sent back to your repository). IMPORTANT: please make sure to delete all the test reports in your final submission.

## Report

You must write a report (named report.pdf) documenting your implementation, displaying your results in a meaningful way, and analyzing your findings, for each part of the assignment. In your report, you should present your implementation, the experimental setup and results. You should discuss what you noticed, draw conclusions and analyze the tradeoffs, if any. The report should be written in a scientific manner (clear structure, clear description of your approach, results, findings, etc., and should use technical writing instead of colloquial terminology or phrases). Keep in mind that presenting your experimental findings and observations to a technical audience is an important skill to develop as a computer scientist.

## Submission

You must keep the same structure in your repository as the starter_code folder from starter.tgz (i.e. it should contain the join.c, join-omp.c, Makefile, etc. at the top level). You must submit all the files required to build and run all your programs (including any header files and the makefile, etc.) except for the datasets, which you must not include. You shouldn't need to add any new files to the starter code, but if you do, make sure that you update the makefile accordingly. Make sure your code compiles and runs correctly on the lab machines. Be sure to make it clear in the report how to run your code! Do not submit any executables or object files! (do a "make clean" before making your final commit).

IMPORTANT: make sure to keep the optimization options (-O3 -DNDEBUG) in the makefile in your final submission, otherwise your code will be too slow, even if your algorithms are efficient.

Aside from your code, you must submit the report (named report.pdf) documenting your implementation, presenting the results, and discussing your findings. When discussing your approach, feel free to also describe any problems encountered and workarounds, what isn't fully implemented (or doesn't work fully), any special design decisions you've taken or optimizations you made (as long as they conform to the assignment specs!), etc.

Additionally, you must submit an INFO.txt file, which contains as the first 2 lines the following:

your name(s)
your UtorID(s)

If you want us to grade an earlier revision of your assignment for whatever reason (for example, for saving some grace tokens if you had a stable submission before the deadline, tried to add new functionality after the deadline but broke your submission irreparably), then you may specify the git hash for the earlier revision you want marked.

As a general rule, by default we will always take the last revision before the deadline (or last one after the deadline, up to your remaining unused grace tokens), so you should not be including a line with the git commit hash, except in the exceptional circumstances where it makes sense. So in general, please avoid using this option and just make sure that the last revision (either before the deadline if you submit on time, or up to a subset of your remaining grace tokens if you submit late) is the one you want graded.

Finally, you must submit a plagiarism.txt file (in the top directory of the assignment), with the following statement:

"All members of this group reviewed all the code being submitted and have a good understanding of it. All members of this group declare that no code other than their own has been submitted. We both acknowledge that not understanding our own work will result in a zero on this assignment, and that if the code is detected to be plagiarised, severe academic penalties will be applied when the case is brought forward to the Dean of Arts and Science."

A missing INFO.txt file will result in a 10% deduction (on top of an inherent penalty if we do not end up grading the revision you expect). Any missing code files or Makefile will result in a 0 on this assignment! Please reserve enough time before the deadline to ensure correct submission of your files. No remark requests will be addressed due to an incomplete or incorrect submission!

Again, make sure your code compiles without any errors or warnings.
Code that does not compile will receive zero marks!

## Marking scheme

We will be marking based on correctness (50%), coding style (10%), and report (40%). Make sure to write legible code, properly indented, and to include comments where appropriate (excessive comments are just as bad as not providing enough comments). Code structure and clarity will be marked strictly! Once again: code that does not compile will receive 0 marks! More details on the marking scheme:

Sequential join: 10%
Shared-memory parallel (OpenMP) join: 40%
Report: 40%
Code style and organization: 10% - code design/organization (modularity (if applicable), code readability, reasonable variable names, avoid code duplication, appropriate comments where necessary, proper indentation and spacing, etc.)
Negative deductions (please be careful about these!):
Code does not compile: -100% for *any* mistake, for example: missing source file necessary for building your code (including Makefile, header files, etc.), typos, any compilation error, etc
No plagiarism.txt file: -100% (we will assume that your code is plagiarised and that you wish to withdraw your submission, if this file is missing)
Missing or incorrect INFO.txt: -10%
Warnings: -10%
Extra output: -20% (for any output other than what is required in the handout)
Code placed in other subdirectories than indicated: -20%
Submitted unnecessary files (compiled code, test reports, data files, etc.): -10%