Software Development for Information Systems (2019 – 2020)

Antonis Klironomos

Chrysostomos Rampidis

Fotis Ektoras Tavoularis

1. Introduction

The following study was executed as part of the course on software development for information systems in the *Notional and Kapodistrian University of Athens*. The aim of this study was to create a subset of a database that manages data entirely in the main memory. The whole task was split in three parts. In the first part we implemented a Sort Merge Join Algorithm, in the second one we performed a query analysis, similar to the one used in the *SIGMOD 2018* competition, while in the last part we implemented CPU personalization and a query optimizer. The project is hosted in GitHub (https://github.com/hectortav/Project2020) and contains a Makefile for compilation and a bash script (testProgram.sh) to easily test the program. To run the executable, produced by the Makefile, called "final", you can use one or more of the following flags to provide cli arguments:

- -qr (QueRy) Run queries of every batch in parallel
- -ro (ReOrder) Run bucket reorder (radix-sort) in parallel
- -pb Create a new parallel job for each new bucket
- -qs (QuickSort) Run quicksorts independently
- -jn (JoiN) Run join in parallel
- -ft (FilTer) Run filters in parallel
- -pj (ProJection) Run Projection checksums in parallel
- -all (ALL) Everything runs in parallel
- -n "threads" Specify number of threads to run
- -jnthreads Complementary way to manage Join

2. Observations and choices during implementation

We implemented and tested two different methods for sorting buckets during the first and second part. One of them was using recursion while the other one was using a loop to break the buckets into smaller ones when needed. After thorough testing we decided to use the recursive one since we managed to achieve better execution times and convolution while we also managed to keep each thread work on a different memory part at any time, reducing the need for semaphores which would slow the execution further since each thread would have to wait to access the memory.

Another part of the execution that we tried to parallelize was the copy from one tuple to another during each call of the Reorder function (tuplereorder_parallel @ functions.cpp), but we finally decided not to keep the change since it significantly slowed the program because all the extra threads that would be used to execute the copy could be used more efficiently by another function that is waiting in the scheduler.

To adapt Join for running on a parallel processing system, we tried several different implementations. First, we tried the one described in class, that is to join areas that have the same prefix. Then, we tried merging some of these areas together so that each new area cluster is assigned to a thread. Finally, we tried to split the relation in equal parts so that each part is assigned to a thread. We noticed that the third method that kept the job count at a minimum while at the same time accomplished a balanced load between threads, produced a better result concerning time and processor usage.

We noticed that we had great time improvements when we used smaller and continues memory allocations. For example by turning histogram and psum from 2D arrays to 1D we managed to cut the execution time to 20% of what we had with the model described in class. The reasoning behind this great improvement is that a big part of the array can be kept in cache and we don't have to worry about loading times that take a long time.

Another observation was the difference between the small and medium inputs. We noticed that the multi threading was better exploited with the medium sized input, where we saw greater time and CPU usage improvements. The explanation is that the time that it takes to split the jobs and wait for an available thread is better justified when the inputs are greater and the time relation of the thread organization processes to the time of the rest of the processes tend to zero.

While running the program through a profiler, we noticed that partitioning during predicate optimization, consumed 37% of the overall time. That means there are many data in a small number of buckets. That way we do not have the optimal outcome we could accomplish with an other optimization policy.

Another observation we made is that the list functions (e.g. insert) consume 17% of the time. If we had another data structure that would not require constant checks and processes to allocate new memory blocks we would achieve

better times with probably an allocation of unneeded memory as a negative aspect.

3. Time Statistics

The following table contains the program runtime measurements, CPU and memory usage with various parameters:

3.0.1. *Small Sized Input*

	1 thread (serial)	2 threads	4 threads	8 threads	16 threads	32 threads	64 threads
No predicate optimization	0.775	N/A	N/A	N/A	N/A	N/A	N/A
Predicate optimization	1.111	N/A	N/A	N/A	N/A	N/A	N/A
(No predicate optimization) Parallel queries (No preddicate optimization)	N/A	0.856	0.662	0.645	0.671	0.671	0.727
Run the following in parallel (Radix-sort, Quicksort, Join, Filters, Projection) (No predicate optimization)	N/A	0.841	0.837	0.856	0.890	0.914	0.906
Everything runs in parallel (Queries, Radix-sort, Quicksort, Join, Filters, Projection)	N/A	0.889	0.806	0.744	0.719	0.737	0.785
(Predicate optimization) Parallel queries (Preddicate optimization)	N/A	1.124	1.016	1.017	0.999	1.061	0.998
Run the following in parallel (Radix-sort, Quicksort, Join, Filters, Projection) (Predicate optimization)	N/A	1.050	1.016	0.976	0.983	0.974	0.978
Everything runs in parallel (Queries, Radix-sort, Quicksort, Join, Filters, Projection)	N/A	1.234	0.895	0.832	0.775	0.822	0.854

Table 1.

Table 2. Time in seconds

3.0.2. *Medium Sized Input*

	1 thread (serial)	2 threads	4 threads	8 threads	16 threads	32 threads	64 threads
No predicate optimization	47.211	N/A	N/A	N/A	N/A	N/A	N/A
Predicate optimization	48.637	N/A	N/A	N/A	N/A	N/A	N/A
(No predicate optimization) Parallel queries (No preddicate optimization)	N/A	44.903	24.862	20.573	20.471	20.844	21.023
Run the following in parallel (Radix-sort, Quicksort, Join, Filters, Projection) (No predicate optimization)	N/A	41.879	38.320	37.520	38.142	36.670	38.874
Everything runs in parallel (Queries, Radix-sort, Quicksort, Join, Filters, Projection) (Predicate optimization)	N/A	47.957	35.500	29.447	20.235	20.767	22.356
Parallel queries (Preddicate optimization)	N/A	45.596	23.769	20.072	20.433	20.597	20.876
Run the following in parallel (Radix-sort, Quicksort, Join, Filters, Projection) (Predicate optimization)	N/A	40.381	37.549	36.599	36.317	36.327	36.604
Everything runs in parallel (Queries, Radix-sort, Quicksort, Join, Filters, Projection)	N/A	47.481	35.455	28.007	18.521	19.425	21.666

Table 3.

Table 4. Time in seconds

---

4. Parallelization Observations

At the third and last part of the project, parallelization was added to the program. The effect this transition had on the program is immediately apparent when looking at the overall execution times. Adding CPU parallelization led to a decline of 20% concerning execution time and a increase of 70% concerning CPU usage which means computer resources are better utilized and the overall outcome is maximized.

5. Version Control Observations

During the whole project we used git as a version control system. With git we managed to track bugs that emerged during development and then easily solve them. Git also helped our team to share code and communicate better that led to better organization and management. Finally, a git aspect we found greatly useful was branching since each team member could work in separate features without causing problems and interference.

6. Unit Testing

To test the execution of our program we used unit testing and especially Cunit. We tried to make unit tests for every function regardless of the size or simplicity. Some of the functions tested are the following:

- randomIndex
- swap
- hashFunction
- makeparts
- splitpreds
- optimizepredicates
- predsplittoterms
- sortBucket
- histcreateTest
- psumcreateTest
- tuplesReorderTest
- InputArray::filterRowIds
- InputArray::extractColumnFromRowIds
- IntermediateArray::populate
- IntermediateArray::findColumnIndexByInputArrayId
- IntermediateArray::findColumnIndexByPredicateArrayId
- IntermediateArray::selfJoin

Overall we concluded that unit testing in an easy way to test and validate the solidity of a big project, that would otherwise require many hours of manual testing. Also we found that creating unit tests before the project is a valid way to ensure the smooth development of a program as well as keeping the team in track and necessitate the usage of a common API.

7. Conclusion

During this assignment we used various technologies and researched a plethora of ideas and theories. We appreciated concepts like parallel programming, unit testing, version control and of course software development for information systems. Parallel programming is a useful asset today since all systems have multiple real and virtual CPU cores that are not often utilized correctly. Unit testing is a respectful way to ensure that programs & algorithms are valid and ready for production and version control is an easy way to track bugs and work with teams. Information systems and databases are the core of today's computing and leaving in an age where data are abundant, we must explore new and more efficient ways to store, find and index them. Last but not least, we learned the importance of pre-planning, estimating possible obstacles and chances during development of such last projects.