A Study of Partitioning Algorithms For In-Memory Databases

IT-University of Copenhagen

Tobias Skovgaard Jeppesen
tobj@itu.dk

Bachelor Thesis
BIBAPRO1PE

May 15th 2019

Supervisors

Phillippe Bonnet
phbo@itu.dk

Pinar Tözün
pito@itu.dk

Abstract

The introduction of multi-core hardware around the 2002 and large tech companies pushing the development of hard analytical workloads on big data, has a large impact on the software design for modern hardware.
In this project I hightlight some of the challenges of adapting software to perform optimally on server hardware. I study, implement and evaluate several partitioning algorithms, to help utilize the multi-core parallism of hardware.

Contents

1 Introduction

In 1965 Gordon Moore made the observation that the number of transistors in intergrated circuits doubled every year[1]. For the remaining years of the 20th century it looked like CPU clock speeds would follow the same trend with an increase of around 50% per year, up until 2002 where the clock speeds have virtually plateaued. [1]

The power consumption of a processor is proportional to the clock speed. For this reason, CPU manufacturers have have hit a wall when it comes to increasing clock speeds due to the challenge of dissipating the heat. Moore's law continues to hold true, so where do the transistors go if CPU's aren't getting faster anymore? Computational throughput is still being increased. The transistors are however used for multiple processor units on the same chip.

To utilize additional cores on modern chips, software has to be redesigned to run in a concurrent or parrallel fashion. High-performance online services(Twitter, facebook, etc.) and big data analytics are dependent on large data-management systems. For the intense workloads that these systems have to sustain, it is vital to write software that utilizes all of the hardware resources.

In large data management systems it is essential that queries and analytical workloads are executed as effectively as possible. To achieve higher performing database systems, it is necessary to program the systems with the intended hardware in mind. Most of the hardware produced today is designed for parallel computing, therefore to utilize the resources offered by modern hardware, the software has to comply.

For large data applications, it is advantageous to partition data into smaller segments, to be individually processed by different processor cores. Thus dividing the workload for parallel computing. The goal for partitioning algorithms is to exploit the memory hierarchy effectively, to relocate data for the hardware.

For this project I will implement partitioning algorithms and evaluate their performance on server grade multi-core hardware. I will thoroughly run tests to evaluate the scalability on multi-core hardware, and other factors that affects the runtime of multiple partitioning algorithms.

2 Background

2.1 Multi-core Scalability

Each year, new processors are released with increasing number of cores. As stated in [2] modern hardware continues to offer increasing parallism. Hardware is evolving with more and more parallel capabilities and features. Utilizing these features to their fullest has always been a constant challenge for software.

It's increasingly difficult to design software to utilize the high amount of the thread-level parallism incorperated in multi-core machines today.

[1] Gordon Moore revised the prediction to doubling every two years in 1975

2.1.1 Implicit Parallism

Before 2005, hardware mainly evolved in the area of single threaded performance within a core. This is most simply done by increasing the clocking of the core, thus doing more cycles in less time. Performance has also been increased with architectual changes and features, such as long execution pilelines and SIMD support.

Implicit parallism refers to this instruction level parallism that is implemented in a core. [2]

2.1.2 Explicit Parallism

Since 2005, multi-core machines have become very common-place in server environments. The increase of possible cores on a CPU has added a new dimension to hardware level parallism. Older software that was designed with single core hardware in mind now has to be redesigned to utilize the computer resources of multi-core machines. The number of cores in a machine can be further increased with multi-socket hardware.

Explicit parallism refers to the parallism of multi-core hardware. Software today has to scale effectively across cores to avoid underutilizing hardware.

2.2 Partitioning

Partitioning is a process of taking a large dataset and dividing it into smaller workable segments. For many server workloads today, the entire dataset worked on is able fit in RAM. In the context of modern hardware, partitioning is often done to distribute workload among CPU cores or CPU sockets. Partitioning is a step to assist database workloads and queries to utilize the parallel capabilities of the hardware. Thus making the memory bandwidth as the main bottleneck of many database tasks.

Partitioning is useful for operations such as joins, aggregations and sorting. The data table is divided into smaller pieces based on the keys of each data entry. Thereafter each piece can be processed by each core individually and in parallel.

Different types of partitioning are distingiushed based on their *partition function*. The *partition function* is what defines what partition a key belongs in. [3] describes 3 types of partitioning, namely *Hash*, *Radix* and *Range*.

Hash and radix have the advantage of being quick operations while range takes longer to compute for each key. One advantage of radix is that the partitioned dataset will be partitioned from low to high by their key value, which makes it useful for sorting. However, radix partitioning is rather susceptible to data scew. In [3] Orestis presents a SIMD-accelerated range partition function that can compete with hash and radix for sorting operations. Because range partitioning is comparison based it accomodates scewed data much better.

2.2.1 Non-in-place & In-place

When considering partition algorithms they are usually implemented either In-place or Non-in-place. Out-of-place partitioning will output the partitioned data into a seperately allocated table. Thus allocating extra memory linearly to the size of the table. In-place algorithms are implemented such that all operations reside inside of the existing table, only using swap operations to move data around. In-place algorithms won't need to allocate more memory in relation to the size of the table.

2.2.2 In-cache & Out-of-cache

In the area of data intensive operations, such as partitioning, memory stalls are the main concern for good performance.

In-cache *In-cache* partitioning algorithms are designed for data sizes that fits in the faster cache memory of the CPU. This means that these algorithms are less concerned about capacity misses as the data fits in the cache already.

Out-of-cache Once data exceed the size available in cache there are more factors that might slow down execution. *Out-of-cache* algorithms are designed to avoid cache polution. These algorithms utilize a buffer to schedule the reading and writing to main-memory. This keeps the cache less poluted and decreases the number of cache misses caused by random writes.[3]

When the dataset is larger than the cache size, data has to be loaded and stored to/from the cache as data is worked on. Thus some capacity misses are ineviteable.

If care is taken around how capacity misses happen we can achieve a faster program. Making sure to access data sequentially is a good strategy to reduce memory stalls. Modern hardware utilizes pre-fetching techniques to predict what data a program will access next. If it's possible to rewrite a program to only access data sequentially in memory the CPU will spend less cycles waiting for data to arrive from memory. However not all computational tasks can be made sequential.

Partitioning can not be made with sequential access to memory only, due to the somewhat undpredictable nature of the partition-functions. For out-of-place partitioning, tuples can at least be loaded sequentially. As the number of partitions P gets larger, there are more possible positions in memory that each tuple could be written to. For example, if a tuple is written to partition 0 or partition 44 will seem random to the CPU. Random acceses are bad for cache memory performance and will clutter the cache.

Data is loaded from memory into cache in blocks called *cachelines*. Different machines can have different sizes of cache lines. If a machine has a cacheline size of 64 bytes, it will hold eight 64bit integers in each cacheline. When a cache-line is loaded into cache and expelled only to access one of the eight integers, the amount time taken to load an entire cacheline is "wasted". Even worse if

the same cacheline has to be loaded again later to access another of the eight integers. Instead a programmer could design their program to use all the eight integers once they are loaded into cache. Thus avoid having to load and expell the same cacheline multiple times.

For large datasets, memory stalls can also be caused by the delay of TLB misses. The TLB[2] works like a cache for the page table. When the data table is large enough, each partition may have to be on different pages. The TLB might not have the capacity to keep page entries for each partition. A machine might have a TLB that contains 32 entries. If the number of partitions P is larger than 32, then each time a tuple has to be written to a partiton that is not in the TLB, the CPU might have to wait for a page walk. Orestis [3] presents out-of-cache algorithms to decrease the stalls from TLB misses.

2.2.3 Shared nothing & shared segments

Partitioning in parallel can either be working on shared segments of data or in a shared-nothing fashion. Shared partitioning is when a number of threads coorporate to partition the same segment of data. This requires some synchronisation between threads as partitions are written concurently. Shared-nothing partitioning requires no synchronisation between threads as threads work seperately on each of their own segments. However with no synchronisation, the resulting partitions will be placed disjointed from eachother between threads.

2.2.4 NUMA boundaries

Larger server environments may employ multiple processors deployed on multiple sockets or clusters. This raises a concern about how memory accesses across sockets or clusters are managed. Convential machines with one CPU has uniform access to memory. In server environments, machines may have several CPUs working together, sharing memory. In this case different areas of memory may have different bandwidths to certain CPUs, meaning they have "Non-Uniform Memory Access". In this case programs can be made NUMA-aware, to account for the different latencies for different areas of memory.

2.3 Concurrent programming with threads

Programming for multi-core hardware is best done with the use of threads. Threading is an execution model for parallel execution.

2.3.1 POSIX-Threads

The POSIX standard defines an execution model for parallel workflows using threads. This is commonly refered to as **pthreads**. The POSIX-thread API support the creation and control of threads.

[2]Translation Lookaside Buffer

For C programming the functions, types and constants to work with threads are defined in the pthread.h libary. When compiling with gcc the pthread library can be linked with the −pthread argument. A list of features offered by the pthread library can be found in resource: [4].

2.4 Thread Synchronisation

Using the functions implemented pthread.h threads can be created, detached and killed. Once a thread is created it will continue working, until it either finishes and kills itself or it is stopped by a call from another thread.[5]

As a thread is done with it's work, it usually needs to be synchronised again at some point. If the caller needs to wait for thread to finish it's it can be joined with the caller thread, using pthread_join().

Condition variables can be used if a thread has to wait to be activated. If `Thread_A` needs to finish something before `Thread_B` continues, conditiona variables can be used so `Thread_B` will wait for a signal from `Thread_A` before `Thread_B` continues. Thus `Thread_A` is responsible to wake up `Thread_B` when it is ready to execute it's workload. Condition variables can be used to wake one thread at a time, or many threads at once.

Barriers come in handy for tasks where multiple threads are working to finish the same job. If a job consists of multiple steps, barriers can be used to synchronise threads between steps. When threads are working on a task in parallel[3], the workload might be scewed, or other factors might delay threads. To avoid `Thread_A` starting on step 2 before `Thread_B` has finished step 1, a barrier can be used to sync all threads between steps.

2.5 Shared Resources

If multiple threads are accessing the same data in memory race conditions may occur and result in undesirable behavior. For example as two threads are attempting to increment the same counter they will race to read and write their results to memory.

1. `Thread_A reads counter` $c \rightarrow 0$
2. `Thread_B reads counter` $c \rightarrow 0$
3. `Thread_A increments counter` $0 + 1$
4. `Thread_A writes new value to counter` $c \leftarrow 1$
5. `Thread_B increments counter` $0 + 1$
6. `Thread_B writes new value to counter` $c \leftarrow 1$

To avoid this pthreads offer features to schedule the access to data for threads.

Mutexes are variabled that can be used to lock access from other threads. If `Thread_A` locks a mutex, `Thread_B` will have to wait for `Thread_A` to unlock the mutex again before `Thread_B` can aquire a lock on the same mutex. This makes it possible to serialize access to shared variables. However if threads have to stall

[3](Concurently, but in a parallel fashion)

to access variables, it may hurt the peformance of the application. Therefore the less resources that threads have to share, the better the parallel scaling will be, generally.

Alternatively *atomic functions* can be used for synchronising simple data changes [4]. Atomic functions are made to safely change the values of variables accross threads, without the need for programming mutex locks.

2.6 Summary

Software has to continually adapt to the evolvment of hardware to utilize the available resources. This requires software designed to scale implicitly and explicitly. Multi-core hardware continues to support more, and more thread-level parallism. Partitioning is a vital tool in data management application to distribute work evenly accross cores in server hardware.

[4] `https://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Atomic-Builtins.html`

3 Experimental Framework

3.1 Data

The data worked on in this project consists of tuples with 64 bit keys, and 64 bit payloads. These tuples are placed in one contiguous array. The data is defined in the source file init.h where the functions for generating, outputting tuples and other accessory functions are declared.

The function:

```
void init_data(tup64* T, uint64_t N_tuples, unsigned int seed);
```

Will generate N tuples with 64bit random keys and store them in T. This function is used to generate all data used in the experiments. The keys are generated using a Mersenne Twister implementation in Random.c[5]. Thus resulting in uniformally distributed keys.

3.2 Histograms

The histogram is an array describing the size of each partition p. All partitioning algorithms described in this paper requires a histogram. The histogram is needed to predict where each partition segment starts. Knowing the size of each partition it's possible to place the tuples contigouously when partitioning. The interface for making the histograms is declared in source file histogram.h. The process of counting a histogram is shown in algorithm 1.

Algorithm 1 Histogram Counting

```
1: histogram ← [0_1, 0_2, ..., 0_P]          ▷ P numbers of zeros
2: for i ← 0 to |T| − 1 do                    ▷ T : tuple table
3:     t ← T[i]
4:     p_0 ← f(t)                             ▷ f : partition function
5:     histogram[p_0] + +                     ▷ Increment bin p_0
6: end for
```

The histogram.h implementations each offers three different approaches to count histograms.

- histogram_serial (...) counts all tuples in one thread.
- histogram_parallel (...) counts all tuples in $N_{threads}$ threads and sums them together to one histogram.
- N_histograms_parallel (...) counts N segments of the tuples with N parallel threads. Returns: N histograms. One for each segment.

There are two implementations supporting the histogram.h interface:

- histogram_hash.c

[5]Copyright (C) 2004, Makoto Matsumoto and Takuji Nishimura, All rights reserved [6]

- histogram_range.c

Each containing their own implementation of the partition function.

3.3 Partition functions

For this project there are two implemented partition functions, namely, *hash* and *range*.

3.3.1 Hash

Hash functions are usually designed to decrease the amount of collisions in hash tables. However in this case, all that is needed is a hash function that distributes keys as randomly or uniformally as possible. [3]

Using B-bit keys, and P is a power of 2. For any tuple t, the partition p is calculated by equation 1. [6]

$$p = (\text{t.key} * m) \gg (B - \log_2(P)) \tag{1}$$

Where m can be any constant odd factor.
In this case it's a factorial prime: $m = 11! + 1$

3.3.2 Range

The range partitioning function is implemented using binary search over an array of delimiters. The delimiters describe what range each partition p extend over. For example, if p_0 is the range $0 - 1000$ and p_1 is the range $1001 - 2000$. Then a tuple t with a key value of 600 will belong to p_1. This results in a partitioning algorithm where all keys in partition p_0 is less than all keys in p_1. This makes range partitioning advantageous for sorting algorithms.

Algorithm 2 Delimiters Binary Search

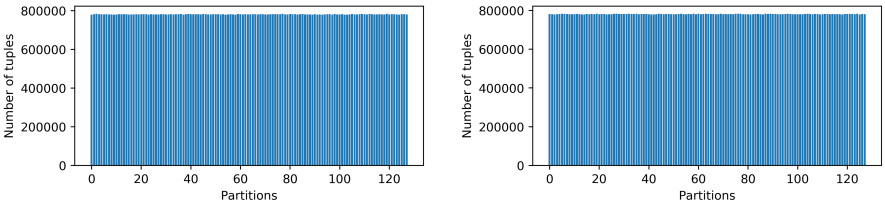
$dl \leftarrow [0, d_{p0}, d_{p1}, \ldots, d_{p(P-1)}]$ ▷ Delimiters
$L \leftarrow 0$
$R \leftarrow (P - 1)$
while $L \neq R$ **do**
  $m \leftarrow ((L + R)/2) + 1$
  if $dl[m] > t.key$ **then** ▷ t.key: The key of tuple t
    $R \leftarrow m - 1$
  else
    $L \leftarrow m$
  end if
end while
return L

[6] In this project B is always equal to 64 bits

Unlike the range function implemented by Orestis [3], the implementation for this project does not utilize the SIMD registers to accelerate the search over the delimiters.
This maks this range partitioning very costly computationally compared to hash partitioning. The search algorithm for range partitioning is illustrated in algorithm 2.

3.3.3 Distribution

The goal of both partition functions is to distribute the tuples as uniformally as possible across the partitions. The distribution of both algorithms on random keys can be seen in figure 1.



(a) Hash histogram (b) Range histogram

Figure 1: Histogram distribution of 10^7 tuples in 128 Partitions

3.4 Partitioning Algorithms

All partitioning algorithms are implemented in `partition.c` as specified in the header file `partition.h` Compiling will produce following binaries:

- `partition_in_serial` ..... In-Place partitioning for in-cache.
- `partition_out_serial` ..... Out-of-Place partitioning for in-cache.
- `partition_out_parallel_a` ..... Shared-nothing parallel out-of-place.
- `partition_out_parallel_b` ..... Parallel out-of-place with pre-synced offsets.
- `partition_out_parallel_atomic` ..... Parallel out-of-place with atomic sync.
- `partition_in_parallel_a` ..... Shared-nothing parallel in-place.
- `partition_in_parallel_synced` ..... Parallel in-place partitioning, shared.
- `partition_in_serial_buffer` ..... In-place with buffer for out-of-cache.
- `partition_out_serial_buffer` ..... Out-of-place with buffer for out-of-cache.

Each algorithm will produce a hash version and a range version.
Out of place
All the out-of-place algorithms use a seperate array to store the output of the partitioning. The input array is only read from, and all partitioned data is written to the output array. This requires more memory of the system, because

the out-of-place algorithms allocates twice as much of memory as the table itself.
In place
In-place algorithms won't allocate memory for their output. Instead all in-place algorithms use swap operations to move tupels to their corresponding partition.

3.5 In-Cache

The in-cache algorithms are designed to partition data segments that are small enough fit in cache.

3.5.1 Out-of-place

The out-of-place is the simplest to implement. It iterates though every tuple in the input table sequentially, and computes the partition number of each tuple. As it iterates, it writes the tuples to the output table at the location of the correct partition. See algorithm 3. The offset array is used to know exactly where each partition begins. As tuples are written to each partition the offset is increased.

Algorithm 3 Partitioning Out-of-place ([3] section 3.1 alg:1)

```
1:  i ← 0
2:  for p ← 0 to P − 1 do                          ▷ P the number of partitions
3:      offset[p] ← i                               ▷ point at the start of each partition
4:      i ← i + histogram[p]
5:  end for
6:  for i_in ← 0 to |T_in| − 1 do
7:      t ← T_in[i_in]                              ▷ T_in : the input table
8:      i_out ← offset[f(t.key)] + +                ▷ f : the partition function
9:      T_out[i_out] ← t                            ▷ T_out : the output table
10: end for
```

3.5.2 In-Place

The In-place algorithm works quite differently. As the input and output reside in the same array, there must be taken more care of how tuples are written so no tuples are overwritten and lost.

For this reason the In-place algorithms only moves tuples around with swap operations. The In-place algorithm runs through a number of *swap cycles*. Each swap cycles starts by picking an initial tuple t_0. The partition that t_0 belongs to is figured out with the partition function. The final position of t_0 is figured using the offsets array. t_0 is then swapped with the tuple in that position. The swap cycles then continues with the tuple swapped with t_0. The swap cycle ends when a tuple t_n belongs in the position t_0 started from. t_n overwrites the initial t_0, and a new untouched tuple is picked to start a new swap cycle. This continues until all partitions have been filled.

Algorithm 4 Partitioning In-place ([3] section 3.1 alg:2)

```
1: i ← 0
2: for p ← 0 to P − 1 do                          ▷ P the number of partitions
3:     i ← i + histogram[p]
4:     offset[p] ← i                               ▷ point at the end of each partition
5: end for
6: p ← i_end ← 0
7: while histogram[p] = 0 do
8:     p + +                                       ▷ Skip initial empty partitions
9: end while
10: repeat
11:     t ← T[i_end]                                ▷ T : the input and output table
12:     repeat
13:         p ← f(t.key)                            ▷ f : the partition function
14:         i ← − − offset[p]
15:         T[i] ↔ t                                ▷ Swap
16:     until i = i_end
17:     repeat
18:         i_end ← i_end + histogram[p + +]         ▷ Skip empty partitions
19:     until p = P or i_end ≠ offset[p]
20: until p = P
```

Orestis describes the in-cache algorithms for partitioning in [3] section 3.1.

3.6 Out-of-cache

To decrease memory stalls with larger datatables, I have implemented out-of-cache partitioning algorithms, as decribed by Orestis in [3]. The out-of-cache algorithms utilize a buffer to store cacheline sized chunks of data for each partition. The buffer is used to schedule writes to the table. Each write to the table is then done one cacheline at the time. The goal of this is to decrease the amount of random writes to memory, thus also reducing the amount of TLB misses. Accessing the different partitions in the buffer will be faster as the buffer will stay in cache in the process of partitioning.

3.6.1 Out-of-place

The out of place algorithm has the advantage that it can still access the input table in a sequential fashion, allowing pre-fetching to reduce memory stalls. There's a cacheline sized buffer for each partition. Once a buffer is full, it is emptied into the output table. Thus the output table is only accessed with one cacheline at a time.

There is a few differences between the implementations for this project and Orestis' implementations described in [3]. See algorithm 5.

Orestis interleaves the offset array into the buffer. This is done in order to save space in the cache for the buffer. In my implementation the offsets are kept separate, similar to the in-cache version, for convenience. This however, clutters the cache more.

The pseudo-code described in ([3] algorithm 3) must assume that each partition has a number of tuples divisible by L. If partition segments aren't cache aligned, some buffers would never be emptied, or be at risk of overriding other tuples.

For these, algorithm 5 includes the handling of partitions that are not cache-line aligned. This requires an additional array, `cacheoffset`, to account for the offset alignment of each partition. The `cacheoffset` array is only needed when emptying the buffer.

An aditional step is needed to empty the remaining tuples left in the buffer after partitioning. See algorithm 5 line 22 to 27.

Algorithm 5 Partitioning Out-of-place with buffer

```
 1: i_out ← 0
 2: for p ← 0 to P − 1 do                                   ▷ Make offsets
 3:     if i_out mod L = 0 then          ▷ If partition p is offset from cachelines
 4:         cacheoffset[p] ← 0
 5:     else
 6:         cacheoffset[p] ← L − (i_out mod L)
 7:     end if
 8:     offset[p] ← i_out + cacheoffset[p]
 9:     i_out ← i_out + histogram[p]
10: end for
11: for i ← 0 to |T_in| − 1 do
12:     t ← T_in[i]
13:     p ← f(t)
14:     i_out ← offset[p] + +
15:     buffer[p][i_out mod L] ← t
16:     if i_out = (L − 1) then                              ▷ Empty buffer of p
17:         for i_buf = 0 to L − 1 do
18:             T_out[i_out + i_buf − (L − 1) − cacheoffset[p]] ← buffer[p][i_buf]
19:         end for
20:     end if
21: end for
22: for p ← 0 to P − 1 do                    ▷ Empty last tuples left in buffer
23:     while offset[p] mod L ≠ 0 do
24:         offset[p] − −
25:         T_out[offset[p] − cacheoffset[p]] ← buffer[p][offset[p] mod L]
26:     end while
27: end for
```

3.6.2 In-place

The In-place out-of-cache (see algorithm 6) uses a buffer similarly to the out-of-place version. With the difference being that the buffers is filled at first, and tuples are swapped between the buffers in-cache.

There is again some differences between this implementation and the one shown by Orestis in [3]. Like the out-of-place version, this implementation stores offsets in a seperate array for convenience instead of interleaving the offsets in the buffers.

The out-of-cache in-place version for this project is quite different from what is described in [3]. Again the implementation for this project also handles partition sizes not divisible by L, while the out-of-cache implementations in [3] does not seem to.

The first steps in algorithm 6 is to count the offsets and fill the buffers. The buffers are filled differently if they are aligned to the cachelines or not. The swap cycles can then run much like they do in the in-cache version, except all swaps are done in the buffers. When all swaps in a buffer is finished the buffer is written to memory and refilled with new tuples, and the swap cycle continues. The swap cycle is interupted if a tuple belongs in the location of the tuple picked initially (see alg 6 line 29). The steps to find a new tuple to initiate a new swap cycle are shown in algorithm 7 line 30 - 36.

When it is no longer possible to find new tuples for swap cycles (alg:7, line 37), it means the partitioning step is over and the buffers have to be emptied for the last time. (See Algorithm 7)

3.7 Parallel partitioning

The algorithms described so far are only described as serial algorithms. However, some can be made parallel with very small changes.

3.7.1 Shared-nothing partitioning

All algorithms in this project can be parallised in a Shared-nothing fashion. This means, instead of multiple threads cooperating on one segment of data, the table is evenly divided into segments so each thread can work on their own segment of data. Threads working in a shared-nothing fashion means no synchronisation between threads is needed during partitioning. The drawback to this is that the resulting partitioning will be disjointed to the number of threads. Partition 0 in `Thread_A` won't be connected to partition 0 in `Thread_B`. Shared nothing parallel partitioning is implemented in the programs `partition_out_parallel_a` and `partition_in_parallel_a`.

3.7.2 Shared partitioning - Non-in-place

The out-of-place algorithms can be made parallel, sharing one segment, by synchronising the offset arrays between threads. The simplest way of synchronising the offsets, is having the offset array be a shared variable among the

Algorithm 6 Partitioning In-place with buffer

```
1:  i ← 0
2:  for p ← 0 to P − 1 do                                   ▷ Count offsets
3:      end[p] ← i
4:      i ← i + histogram[p]
5:      offset[p] ← i
6:      for i_buf ← 0 to L − 1 do
7:          if i mod L ≠ 0 then                              ▷ Fill buffer
8:              buffer[p][i_buf] ← T[i − (i mod L) + i_buff]
9:          else
10:             buffer[p][i_buf] ← T[i − L + i_buf]
11:         end if
12:     end for
13: end for
14: i_end ← 0
15: t ← T[0]
16: loop
17:     repeat
18:         p ← f(t)
19:         i ← − − offset[p]
20:         t ↔ buffer[p][i mod L]                           ▷ swap
21:         if i mod L = 0 then
22:             for i_buf ← 0 to L − 1 do
23:                 T[i + i_buf] ← buffer[p][i_buf]           ▷ Empty buffer
24:             end for
25:             for i_buf ← 0 to L − 1 do
26:                 buffer[p][i_buf] ← T[i − L + i_buf]       ▷ Refill buffer
27:             end for
28:         end if
29:     until i = i_end
                                          ▷ Continued in algorithm 7...
```

threads, and increment the offsets using an atomic function. This way of parallelising the partition on a shared segment is implemented in the program `partition_out_parallel_atomic`. This implementation risks being slowed down as a lot of threads will all be contesting the same variables.

Another, more convenient approach is to synchronise the offsets before partitioning. The offsets can also be synchronised after each threads has counted their histograms. If N number of histograms are counted for N threads for N number of segments. The offsets for each thread can be synchronised to correctly place the partitions from each segment to one large segment. So when the partitioning starts all threads know the correct position to write their tuples without needing to synchronise with the other threads anymore. This is implemented in the program `partition_out_parallel_b`.

Algorithm 7 Partitioning In-place with buffer (continued)

```
30:     for p ← 0 to P − 1 do                    ▷ Find new tuple to initiate swap cycle
31:         if offset[p] > end[p] then
32:             t ← T[end[p]]
33:             i_end ← end[p]
34:             break for                                  ▷ continue partitioning
35:         end if
36:     end for
37:     if i ≠ i_end then                          ▷ If new i_end found: continue
38:         continue loop
39:     else                                                ▷ Otherwise no tuples left
40:         for p ← 0 to P − 1 do                ▷ Empty remaining tuples in buffer
41:             if end[p] mod L ≠ 0 then
42:                 for i ← end[p] to end[p] − (end[p] mod L) + (L − 1) do
43:                     T[i] ← buffer[p][i mod L]
44:                 end for
45:             end if
46:         end for
47:         break loop                                      ▷ Exit partitioning
48:     end if
49: end loop
```

3.7.3 Shared partitioning - In-place

The in-place algorithms are more complicated to sync between threads. Orestis describes an algorithm in [3] to run parallel partitioning in-place on a shared segment. I have attempted to implement the algorithm as described. However in the inner while loop (alg 9 line 14), i is described in ([3] alg 5) to be $i \leftarrow \text{used}[p]$, but the algorithm will not work unless it's changed to $i \leftarrow \text{used}[p_{next}]$

The only two shared resource between threads is the `used[]` array and the collection of addresses for deadlocked tuples $T_{deadlock\_index}$. The threads synchronise the offsets in `used[]` using the atomic function fetch_and_add(c , i)[7]. The `used[]` array and $T_{deadlock\_index}$ is initalized before the threads are spawned. The offsets can also be counted prior to creating threads. (See algorithm 8.

The process of each thread can be seen in algorithm 9. Because the threads are competing for variables in `used[]`, the algorithm checks for "deadlocks" when cycling through swaps. Using the atomic functions there's no chance for an actual deadlock, by the end of a swap cycle, a thread could potentially change the counter in `used[]`, disrupting the work of another thread. This could leave a thread with no space left to write it's current tuple. For this reason there is a safety net for these deadlocked tuples to be cleaned up when the partitioning is over. There will be an index missing a tuple for each deadlock which is logged in the $T_{deadlock\_index}$ collection.

[7] https://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Atomic-Builtins.html

All threads have to finish partitioning before the deadlocked tuples can be fixed. For this reason a barrier is setup on line 27 to synchronise threads.

Orestis mentions in [3] that this implementation may not be practical because a shared variable have to be updated for each tuple moved. There is no in-place shared segment implementation using a buffer to boost out-of-cache peformance either.

Algorithm 8 Partitioning In-place parallel

1: used$[P] \leftarrow [0_1, 0_2, \ldots, 0_P]$
2: $T_{deadlock\_index} \leftarrow [\{\}_1, \{\}_2, \ldots, \{\}_P]$ ▷ deadlock indices for each p
3: $i \leftarrow 0$
4: **for** $p \leftarrow 0$ **to** $P - 1$ **do**
5: offset$[p] \leftarrow i$
6: $i \leftarrow i +$ histogram$[p]$
7: **end for**

8: spawn threads ...

Algorithm 9 Partitioning In-place parallel thread

1: $T_{deadlock\_list} \leftarrow []$
2: $P_{active} \leftarrow \{P_1, P_2, \ldots, P_P\}$
3: **while** $|P_{active}| > 0$ **do**
4: $p \leftarrow$ **any** $\in P_{active}$
5: $i \leftarrow used[p] + +$ ▷ Atomic
6: **if** $i \geq$ histogram$[p]$ **then**
7: $P_{active} \leftarrow P_{active} - \{p\}$
8: **goto** loop-end
9: **end if**
10: $i_{beg} \leftarrow i +$ offset$[p]$
11: $t \leftarrow T[i_{beg}]$
12: $p_{next} \leftarrow \boldsymbol{f}(t)$
13: **while** $p \neq p_{next}$ **do**
14: $i \leftarrow used[p_{next}] + +$ ▷ Atomic
15: **if** $i \geq$ histogram$[p_{next}]$ **then**
16: $T_{deadlock\_list} \leftarrow T_{deadlock\_list}$.append$(t)$
17: $T_{deaclock\_index}[p] \leftarrow T_{deaclock\_index}[p]$.push$(i_{beg})$
18: **goto** loop-end
19: **end if**
20: $i \leftarrow i +$ offset$[p_{next}]$
21: $T[i] \leftrightarrow t$ ▷ swap
22: $p_{next} \leftarrow \boldsymbol{f}(t)$
23: **end while**
24: $T[i_{beg}] \leftarrow t$
25: loop-end
26: **end while**

27: pthread_barrier_wait() ▷ Wait for all threads to finish their swap cycles

28: **while** $|T_{deadlock\_list}| > 0$ **do** ▷ Fix remaining deadlock tuples
29: $t \leftarrow T_{deadlock\_list}$.pop()
30: $p \leftarrow \boldsymbol{f}(t)$
31: $i \leftarrow T_{deadlock\_index}[p]$.pop() ▷ Get free position in p
32: $T[i] \leftarrow t$
33: **end while**

Further info of implementation and source files can be found in appendix A.

4 Experiments

All histogram experiments are implemented in either histogram_range.c or histogram_hash.c, as declared in the header file histogram.h. Compiling will produce the following binaries:

- histogram_serial ..... serial counting of histogram.
- histogram_parallel ..... parallel counting of histogram.
- histogram_n_parallel ..... parallel counting of N histograms.

Each binary will have a hash version and a range version. The experiment's main function is implemented in histogram.c

All partitioning algorithms are implemented in partition.c as specified in the header file partition.h Compiling will produce following binaries:

- partition_in_serial ..... In-Place partitioning for in-cache.
- partition_out_serial ..... Out-of-Place partitioning for in-cache.
- partition_out_parallel_a ..... Shared-nothing parallel out-of-place.
- partition_out_parallel_b .....Parallel out-of-place with pre-synced offsets.
- partition_out_parallel_atomic .....Parallel out-of-place with atomic sync.
- partition_in_parallel_a ..... Shared-nothing parallel in-place.
- partition_in_parallel_synced ..... Parallel in-place partitioning, shared.
- partition_in_serial_buffer ..... In-place with buffer for out-of-cache.
- partition_out_serial_buffer ..... Out-of-place with buffer for out-of-cache.

Each algorithm will produce a hash version and a range version.

The main function that runs the experiments are defiend in partition_test.c. Each binary will allocate and generate N number of tuples. The out-of-place algorithms will also allocate memory for the output tables, before the partitioning step.

All binaries will time themselves and write the runtime in nanoseconds in a file time.t or time_h.t. The timing only includes the partitioning step or the histogram counting. Allocating memory and writing outputs are not included in the experiment results.

Each binary will take arguments for the number of tuples, number of partitions and number of threads.

5 Evaluation and Discussion

5.1 Machine

All experiments are run on the same machine. Specifications for the machine are:

- 2 Intel Xeon E5-2680 v4 2.40GHz 16-core CPUs. Total of 32 cores.
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 35840K
- Total amount of memory: 66GB
- Linux 4.18.0-17-generic - Ubuntu 18.10

5.2 Method

All experiments are measured in nanoseconds. The timing is implemented using the function clock_gettime()[8] which is available for POSIX.1-2001 systems. The time only includes the time spent partitioning. The times does not include allocation of memory.

All timing data is systematically gathered using python scripts. The python scripts will test all binaries with different arguments and allows plotting the data.

All tests are run with tuples of 64bit keys and 64bit payloads. All data is generated as described in section 3.1.

Crontab[9] and Screen[10] has been used to schedule and monitor test runs.

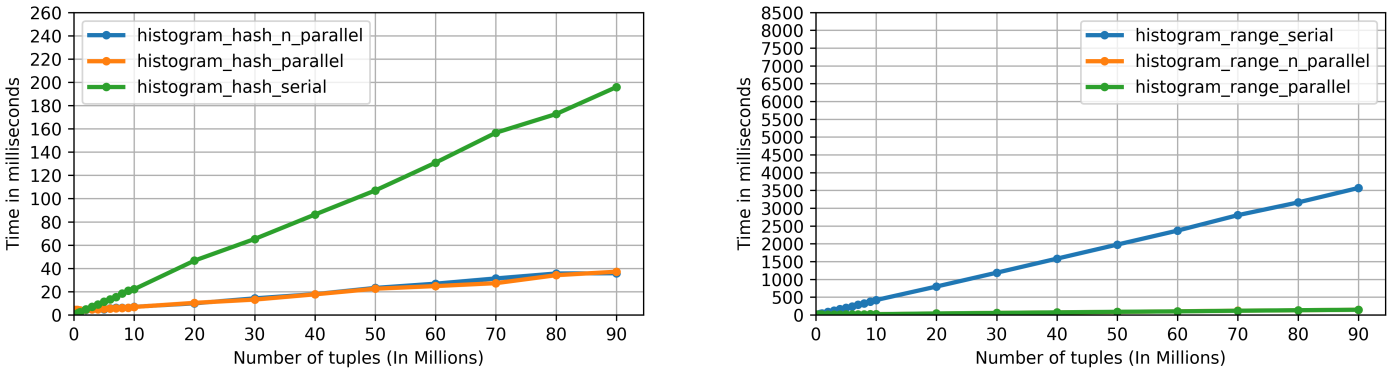
5.3 Histogram

Unless stated otherwise histogram tests are run with the following parameters:

- $N_{tuples} \leftarrow 10^8$
- $P_{partitions} \leftarrow 128$
- $T_{threads} \leftarrow 64$

Each datapoint is the mean of 5 test runs.

[8] `https://linux.die.net/man/2/clock_gettime`
[9] https://www.computerhope.com/unix/ucrontab.htm
[10] urlhttps://linuxize.com/post/how-to-use-linux-screen/



(a) Hash partition function

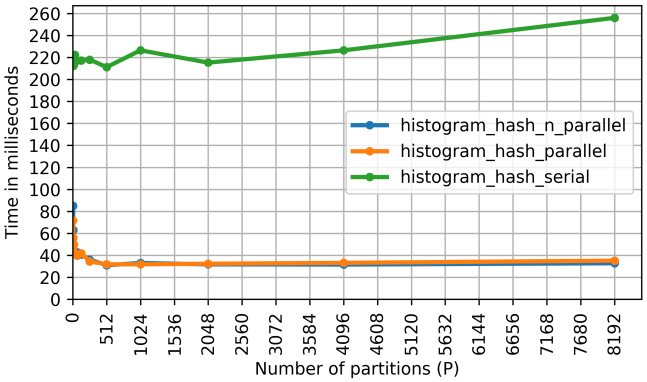
(b) Range partition function

Figure 2: Histogram counting variations, to the number of tuples

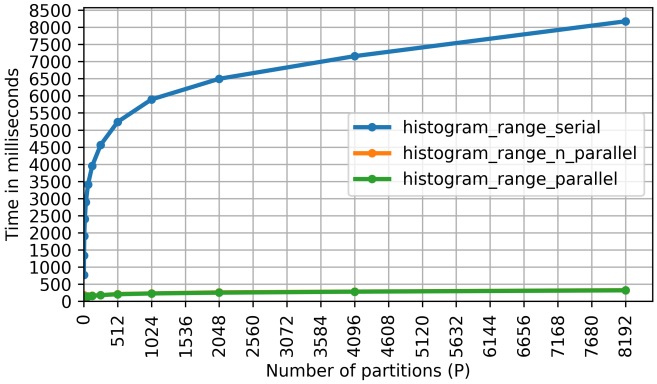
The first thing to notice is the difference in performance between the hash and range variants. The range function is vastly slower due to the very time consuming binary search. The parallel histogram counting runs faster than the serial version as expected on datasets of these sizes. histogram_parallel and histogram_N_parallel run very similarly.

The hash variant in figure 2a, shows very poor scaling compared to the number of available cores (32), while the range variant in figure 2b scales a lot better across threads.

The reason for the differnt scaling could be that hash histogram counting is mainly bounded by the memory bandwidth. The partition function in range histogram counting is much more processor intensive on a small memory area (The delimiters array), so the delays of accessing tuples of the entire dataset is less noticable.



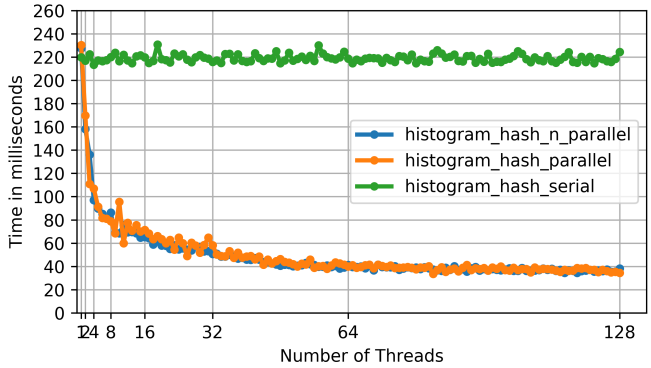
(a) Hash partition function



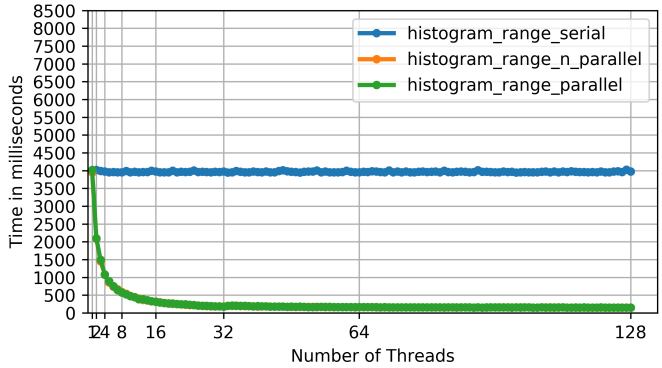
(b) Range partition function

Figure 3: Histogram counting variations with varying number of partitions

Figure 3a and 3b shows that the hash and range functions react differently to the size of P. Figure 3b shows a clear logarithmic trend in regards to P, caused by the binary search algorithm over the P number of delimiters in the partition function. The parallel hash variants show slowdown from lower numbers of P.



(a) Hash partition function



(b) Range partition function

Figure 4: Histogram counting variations with range partition function, with different numbers of threads

The two figures in 4 show how the parallel histogram counting improves as more threads are available. The serial versions stay unchanged as they only utilize one thread. The range histogram in figure 4b show scaling as expected on a 32 core machine. The range histogram performs best with 32 threads and

performance slows down as the number of threads increase from there.

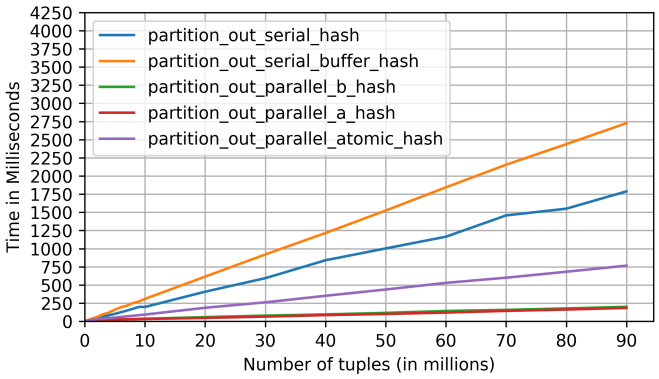
The hash version in 4a shows the same subpar scaling as in figure 2. What is curious, is that the performance continues to improve beyond 32 threads. The higher number of threads helps despite the hardware only having 32 cores.

5.4 Partitioning

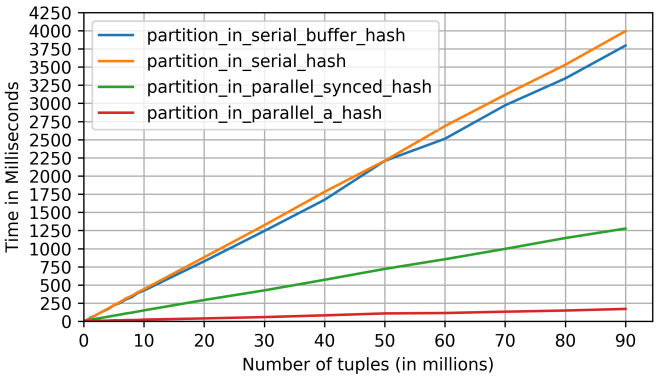
Unless stated otherwise partitioning tests are run with the following parameters:

- $N_{tuples} \leftarrow 10^7$
- $P_{partitions} \leftarrow 512$
- $T_{threads} \leftarrow 32$
- $L_{buffer} \leftarrow 4$

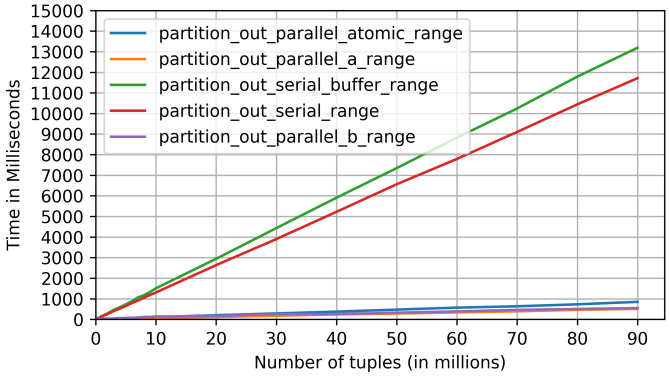
Each datapoint is the mean of 3 test runs. Tests are drawn in groups of hash, range, out-of-place and In-place.



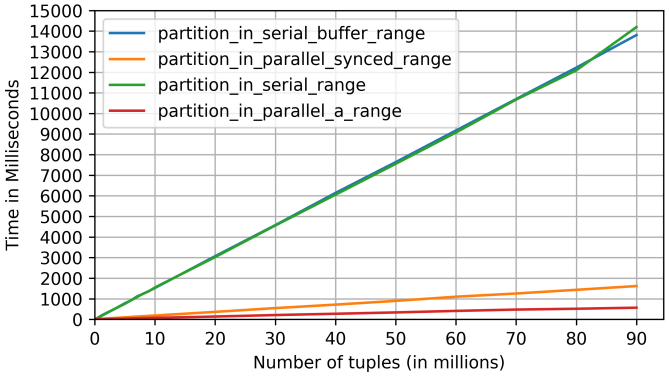
(a) Hash partitioning Out-of-place



(b) Hash partitioning In-place



(c) Range partitioning out-of-place



(d) Range partitioning In-place

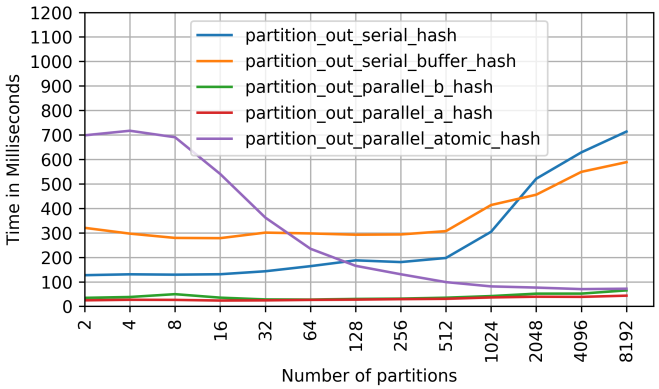
Figure 5: Partitioning to the size of N tuples

In figure 5 the runtime of all partitioning algorithm can be seen in comparison to the number of tuples. There is a lot of things to observe about this data:

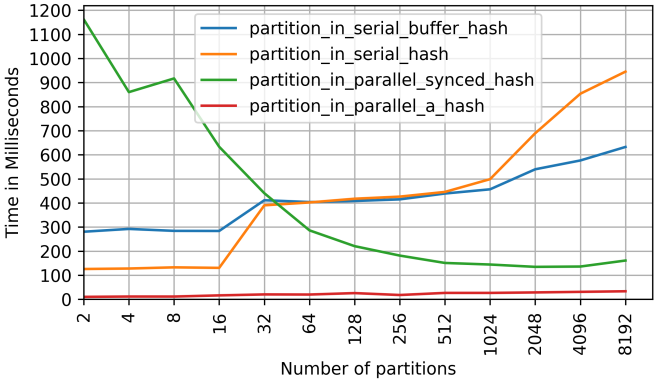
- The hash partition is very much faster than the range partitioning. Much like the data seen in the histogram tests.
- The out-of-place algorithms generally run faster than the in-place ones. However, allocating the output table for out-of-place algorithms is not included in the time data. Notice how in-place compete better with out-of-place versions in the cases of range partitioning compared to hash partitioning.
- All parallel versions are running faster than their serial counterparts, on this scale of data[11].

[11] On small enough datasets, a serial approach would be faster.

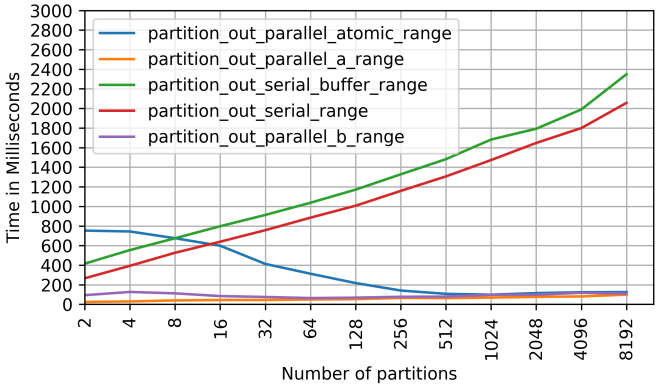
- Range partitioning scales better across cores/threads than hash partitioning does. This is the same behaviour as observed in histogram counting. This again shows that hash partitioning is mainly bounded by the memory bandwidth.
- Algorithms using atomic synchronisation between threads runs much slower than their un-synchronised counterparts. (partition_in_synced and partition_out_par_atomic)
- The partitioning algorithm partition_out_par_b that synchronises offsets before partitioning runs almost exactly as fast as partition_out_par_a that has no synchronisation between threads.
- The buffered approach used in partition_in_serial_buffer shows an improvement compared to the in-cache version.
- The buffered approach used in partition_out_serial_buffer shows more disapointing results with the current parameters.



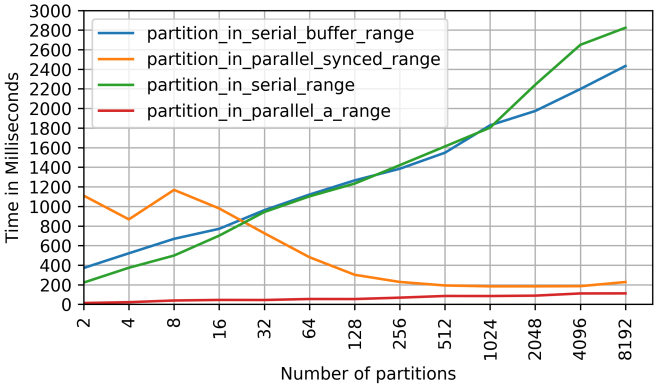
(a) Hash partitioning out-of-place



(b) Hash partitioning In-place



(c) Range partitioning out-of-place

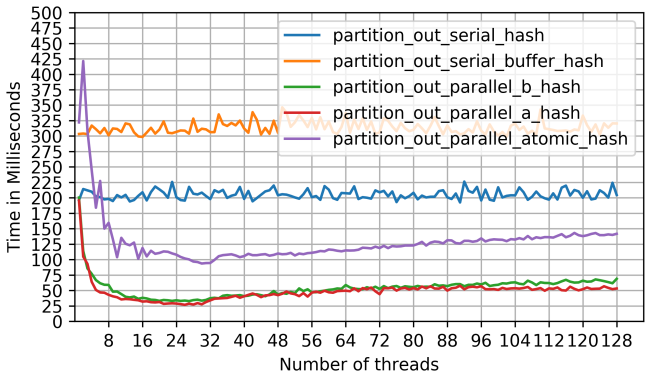


(d) Range partitioning In-place

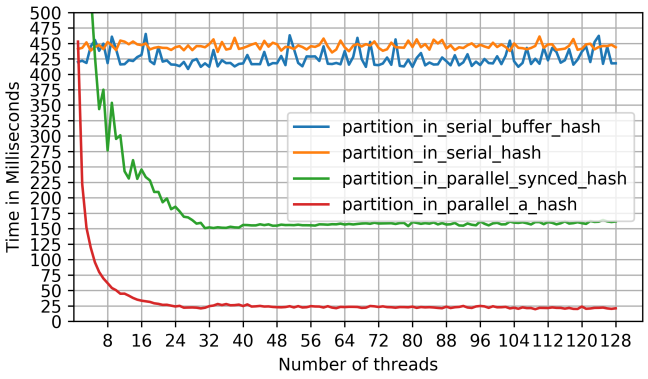
Figure 6: Partitioning algorithms with varying numbers of partitions P

Figure 6 shows the performance in regards to the number of partitions P. Range partitioning is significantly more affected by P than hash is. This is due to the slow range-partitioning runs logarithmically in proportion to P, while hash is constant.

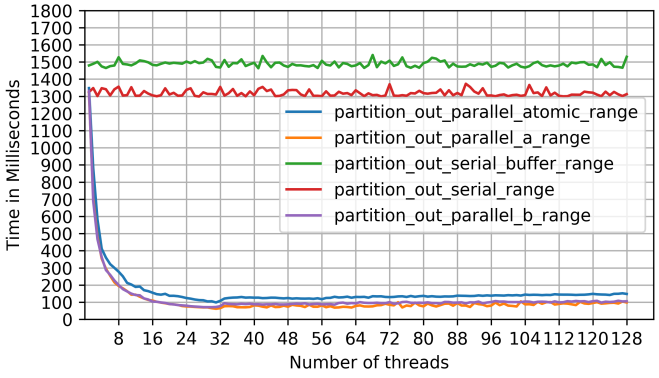
The increase of P reveals that the buffered algorithms performs better as P increases, as expected from section 2.2.2. All parallel algorithms using atomic functions to synchronise threads show better peformance as P increase. With $P > 2^{10}$ atomic synchronisation can almost compete with the non-synchronised versions.



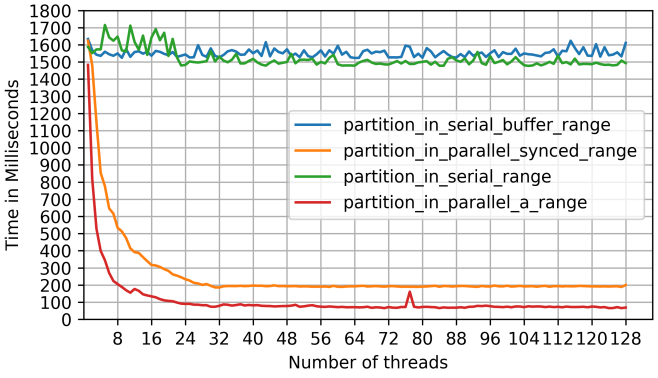
(a) Hash partitioning out-of-place



(b) Hash partitioning In-place



(c) Range partitioning out-of-place



(d) Range partitioning In-place

Figure 7: Partitioning algorithms to number of available threads

Figure 7 shows partitioning algorithms with different numbers of threads available. Ofcourse serial algorithms don't improve with more threads as they will only utilize 1 thread. All parallel algorithms show the best peformance with 32 threads. (Coresponding to the number of available cores)

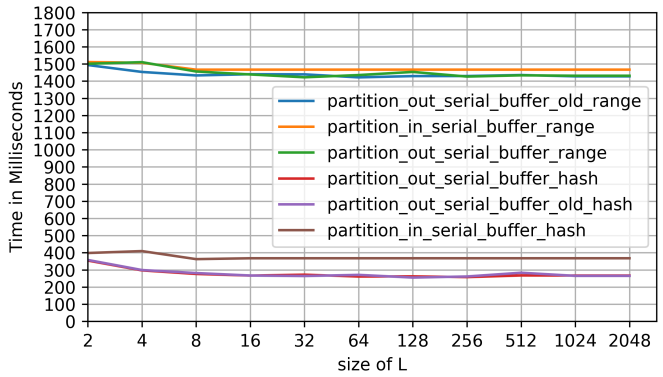


Figure 8: All buffered partitioning with different L sizes for buffers

Figure 8 shows buffered (out-of-cache) algorithms running with different sized buffers. Section 2.2.2 expected that best peformance would be with $L \leftarrow 4$. Data in fig.8 shows that $L \leftarrow 8$ runs faster. The reason for this could have to do with the associativity of the cache, or an unfortunate cache conflict.

Further time breakup of partitions can be found in appendix C.

6 Conclusion

For this project I have implemented 9 different approaches to partitioning large datasets. The project has explored the difference between In-cache, out-of-cache, in-place and out-of-place partitoning algorithms, as well as parallel execution of partitioning. All algorithms are tested on a 32-core machine to evaluate the scalability of the algorithms.

This project has explored the differences between range and hash partitioning. It shows how, range partitioning competes with hash partitioning in runtime.

It shows that the main concern for partitioning large amounts of data is the memory bandwidth and other memory stalls. The evaluation suggest that the performance of out-of-cache partitioning can be improved using buffers for scheduling data read/writes.

The project has offered insight into some details of partitioning large amounts of data. The results of this project can be used to compare different partitioning algorithms, and combining the different variants for specific tasks.

References

[1] Mike Bailey. Parallel programming:moore's law and multicore), 2098. https://web.engr.oregonstate.edu/~mjb/cs475/Handouts/moores.law.and.multicore.2pp.pdf.

[2] Anastasia Ailamaki, Erietta Liarou, Pınar Tözün, Danica Porobic, and Iraklis Psaroudakis. *Databases on modern hardware how to stop underutilization and love multicores*. Morgan & Claypool Publishers, 2017.

[3] Orestis Polychroniou and Kenneth A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. *SIGMOD*, 2014.

[4] Alfred Park. Multithreaded programming (posix pthreads tutorial), 1999-2019. https://randu.org/tutorials/threads/.

[5] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmers Perspective, 3rd edition.* Pearson, 2016.

[6] Takuji Nishimura and Makoto Matsumoto. Mersenne twister 64bit version, 2004. http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt64.html.

Appendices

A Source files

This project includes the following source files:

A.0.1 .h files:

- init.h ..... interface for essential functions such as initializing data
- histogram.h ..... `.h` file for histogram interface
- partition.h ..... `.h` file for partitioning interface
- threadsafe_stack.h ..... `.h` file for a simple stack interface

A.0.2 .c files:

- init.c ..... Implementation of init.h
- histogram.c ..... Main function for runnig histogram programs
- histogram_hash.c ..... Hash implementation of histogram.h
- histogram_range.c ..... range implementation of histogram.h
- partition.c ..... Implementation of partition.h
- partition_test.c ..... Main function running partitioning programs
- partition_time.c ..... Same as `partition.c` but with more time functions
- threadsafe_stack.c ..... Implementation for a stack. (threadsafe_stack.h)
- random.c[12]

A.0.3 .py files:

- test.py ..... Script for timing programs
- test_histograms.py ..... Run all histogram variants with different variables
- test_partitions.py ... Run all partitioning variants with different variables
- test_time_breakup.py ..... Run with time breaks from partition_time.c
- show_histograms.py ..... Plot data from test_histograms.py
- show_partitions.py ..... Plot data from test_partitions.py
- draw_data.py ..... Draw output from partition_test.c programs. (debug)
- draw_histogram.py ..... Draw output from histogram.c programs. (debug)
- HasAllKeys.py ..... Script testing if data is corrupt.

B Global Arguments

The code implemented for this project have a number of global arguments.

- N ..... Number of tuples.
- P ..... Number of partitions.
- L ..... Number of tuples per cacheline.

[12] Copyright (C) 2004, Makoto Matsumoto and Takuji Nishimura, All rights reserved 64bit random number generator [6]

- $N_{threads}$ ..... Number of threads to partition with.

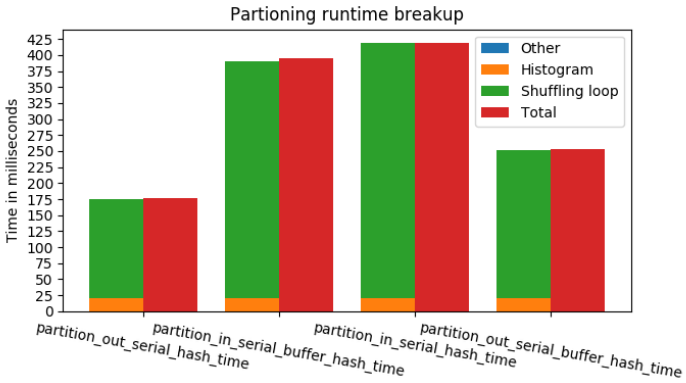
These variables are given as arguments when executing the code. They are declared as **extern** variables in the header files for the modules that need them. histogram.h and partition.h, and should be defined by the caller that uses them (partition_test.c).

C Time breakups of partitioning

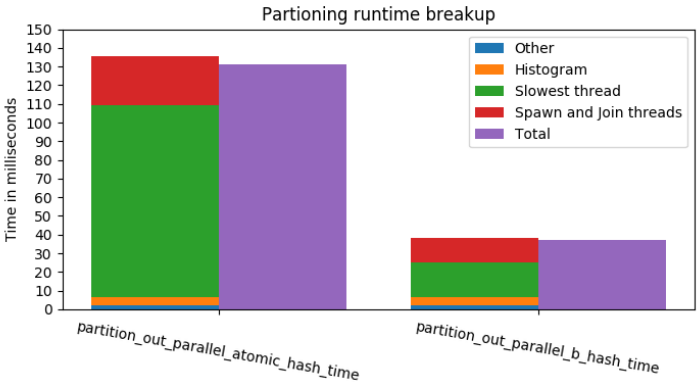
The timing break tests are run with the same default parameters as the other partitioning tests.

- $N_{tuples} \leftarrow 10^7$
- $P_{partitions} \leftarrow 512$
- $T_{threads} \leftarrow 32$
- $L_{buffer} \leftarrow 4$

Datapoints are the mean of 10 runs. The "total" collumn is the runtime of the program without timekeeping of the additional time breaks.



(a) Time breaks of serial algorithms



(b) Time breaks of parallel algorithms

Figure 9: Breakup of partition algorithms