EFFICIENT PROGRAMS

GROUP 14 CONSISTED OF

Milica Aleksic
Ege Aydin
Veneta Grigorova
Thomas Klar
Adigun Oladapo Oludele
Pedro Silva
Christoph Winkler

PROGRAM SPECIFICATION

HIGH-LEVEL DESIGN CHOICES

- Implementation of myjoin project
- Written entirely in C
- Focus on Large input files, potentially ≫12 Mio rows
- Field structure and join order fixed
- Fields can only contain A-Z, a-q, 0-9, \0 characters

ARCHITECTURE

- Data is never entirely loaded into memory
- External merge sorting
- Sort-merge join operations

https://github.com/venetagrigorova/csvmerge

BASE CODE

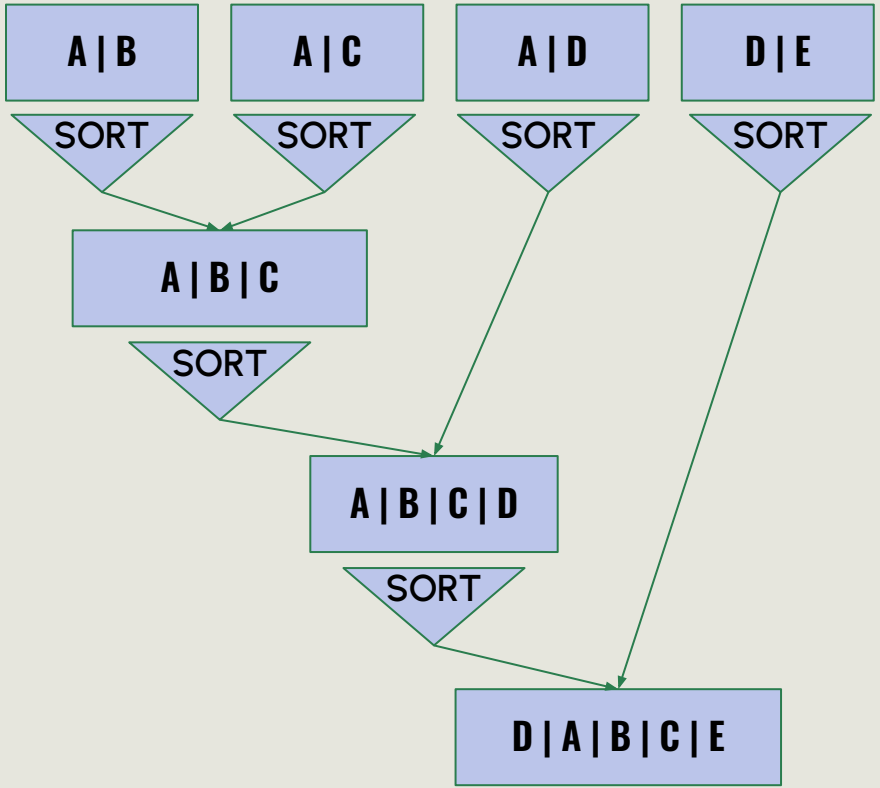
The base implementation performs a sequential 4-way join in the obvious way

DETAILS

- Sequential Chunked merge-sorting
- Record-wise Sort-merge join
- Temp results are written to disk

PERFORMANCE

- ~300 billion cycles, 70 seconds runtime
- Inefficiency distributed, many small function calls accumulate



WHAT IMPROVEMENTS WERE IMPLEMENTED?

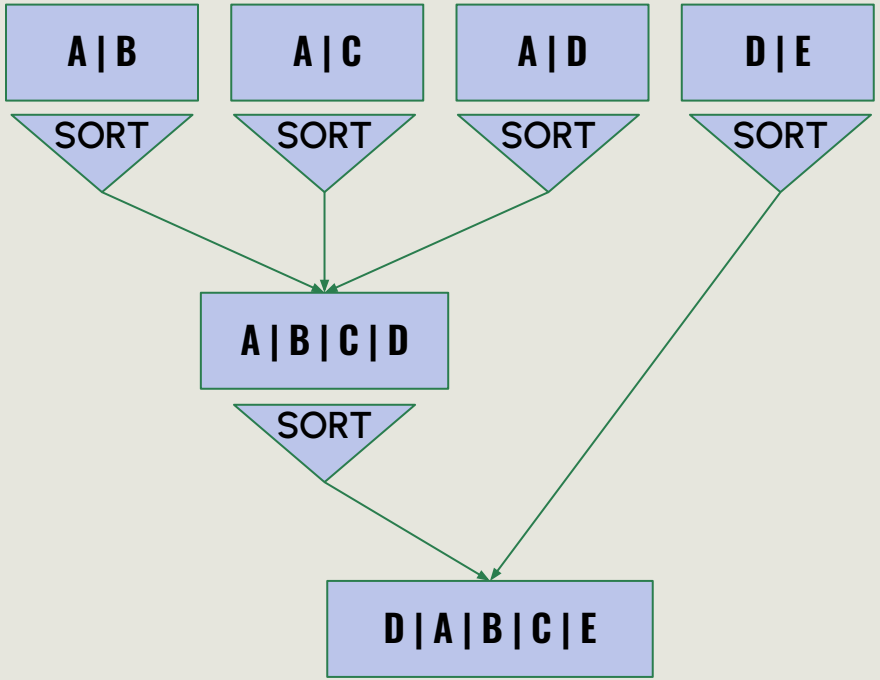
THREE-WAY MERGE JOIN

WHAT CHANGED

- Combined three tables in a single step instead of pairwise joins.

BENEFITS

- Improves performance by reducing intermediate file writes and disk I/O compared to pairwise joins



-19%

INT128 FIELD ENCODING

WHAT CHANGED

- Replace strings with 128 bit Integer numeric encoding
- 22*log_2(56) ≅ 127.76 bits necessary for 56 possible characters
- Encode at start, keep temp files encoded, decode at the very end

BENEFITS / DRAWBACKS

\+ 16 bytes per field vs (up to) 23 bytes
\+ Fixed-length fields
\+ Faster comparisons, copying, access operations
– Requires expensive integer division for encoding and decoding

-33%

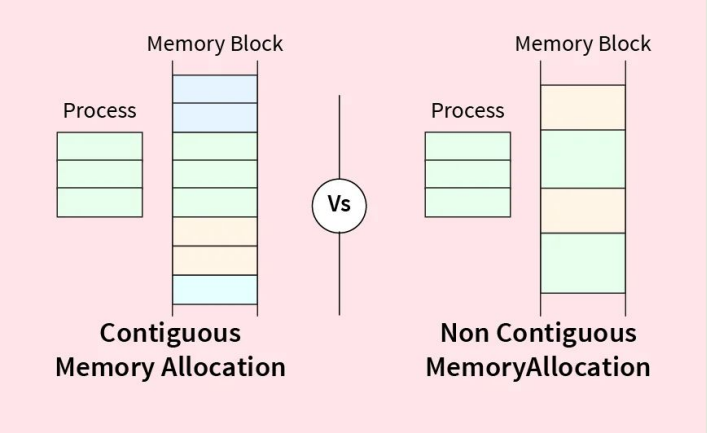
CONTIGUOUS MEMORY STORAGE

WHAT CHANGED

- Stored fields of a chunk continuously in memory (previously only pointers contiguous)

BENEFITS

- Improved cache locality.
- Faster memory access during sorting and joining.



-5%

INTERNAL TABLE BUFFERS

WHAT CHANGED

- Previously: Records are read just-in-time and written to disk immediately after use
- Tables now have internal buffers → prefetching of records
- Keep a fraction of previous records for backwards traversal in join phase

BENEFITS

- Minimizes redundant memory allocations and deallocations.
- I/O operations concentrated and far less frequent
- Improves cache locality by storing records in contiguous memory.
- Eliminates Need for record buffering during join phase

INTERNAL TABLE BUFFERS

BEFORE BUFFER REFILL

Record Index	Field A	Field B
0	Key0	Value0
1	Key1	Value1
2	Key2	Value2
...	...	...
...	...	...
99	Key99	Value99
currentPos	Key100	Value100

AFTER BUFFER REFILL

Record Index	Field A	Field B
90	Key90	Value90
...	...	...
99	Key99	Value99
100	Key100	Value100
currentPos	Key101	Value101
...	...	...
190	Key190	Value190

-6%

SORTING ALGORITHM

WHAT CHANGED

- External merge sort: Sorting and Merging Phase
- In Sorting Phase: Use custom Merge Sort to sort chunks
- In Merging Phase: Use k-way Merge Algorithm to merge sorted chunks

BENEFITS

- Less function overhead
- Merge Sort performs well on highly unsorted data
  - O(n ln(n)) instead of O(n^2)
- K-way Merge reduces number of comparisons (expensive!)

MINOR OPTIMIZATIONS

WHAT CHANGED

- Multiple techniques from lecture applied on critical parts of the code
- Done almost at the end, after algorithmic optimizations

LOOP UNROLLING

- Many iteration steps in loops do not depend on previous steps
- Improves execution speed by enabling compiler optimizations

ARITHMETIC FLAG OPTIMIZATIONS

- Branch reduction and possible compiler optimizations

PRECOMPUTATION OF VALUES

- Precompute Encoding/Decoding of all characters
- Gave very strong performance boost

CODE MOTION OUT OF LOOPS

- A few critical loops have to iterate many times (e.G.. sorting)
- Any saved computation huge win performance-wise

-5%

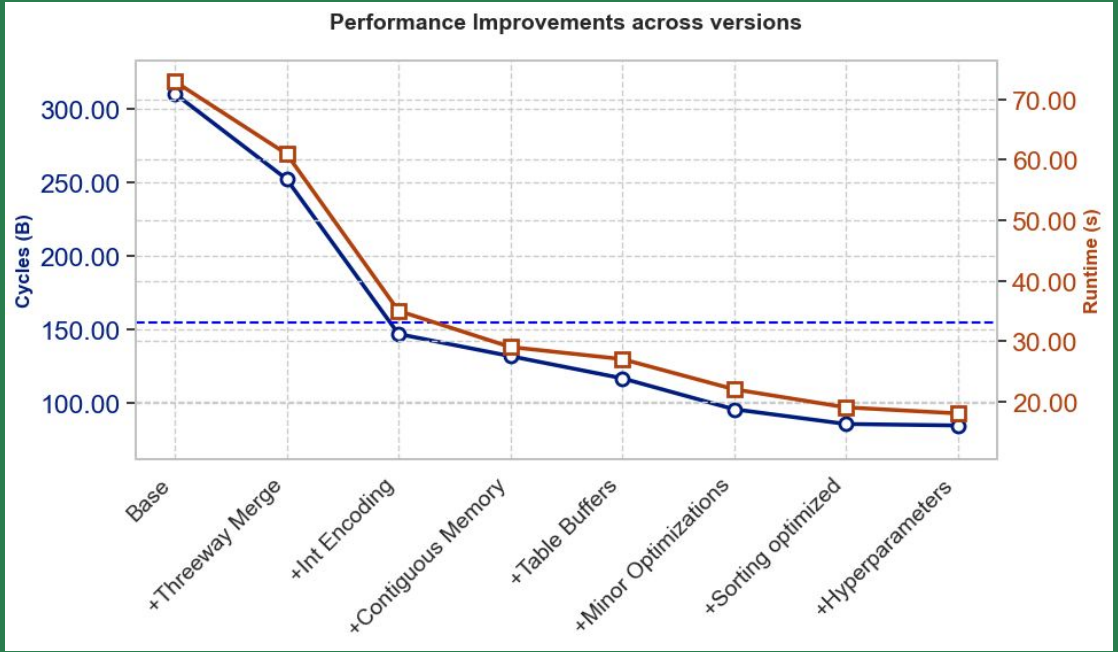
PARAMETER OPTIMIZATION

HYPERPARAMETERS

- Chunk Size/Number of chunks
  - Strong performance at 100.000 Records/Chunk, ≈120 chunks
- Table Buffer Size
  - 4096 Records/Buffer
- Table Buffer History Size
  - 32 Records sufficient
- Optimized experimentally through grid search
- Hard to give theoretical reason for optimal size

PERFORMANCE (CYCLES)

Reference Implementation	Base Implementation	Final Implementation
155,414,886,890 cycles	310,116,076,286 cycles	82,717,600,559 cycles



OTHER PERFORMANCE METRICS

Metric	Reference	Base	Final	Base to Final
Execution Time (s)	39	73	17.9	75% reduced
Cycles user (Billion)	144.97	287.99	72.85	75% reduced
Cycles system (Billion)	15.21	22.88	8.74	62% reduced
Instructions (Billion)	245.25	529.02	174.96	66% reduced
Instructions per Cycle (IPC)	1.51	1.71	2.21	30% improved
Branches (Million)	53,795	123,352	29,979	75% reduced
Branch Miss Rate	2.03%	1.69%	2.47%	50% worse

MULTITHREADING

The g0 machine has 8 active threads → multithreading

CHALLENGES FACED

SYNCHRONIZATION OVERHEAD

- Managing shared resources (e.g., memory buffers, file writes) caused significant overhead.
- Required complex thread synchronization mechanisms (e.g., mutexes), reducing performance benefits.

INCREASED COMPLEXITY

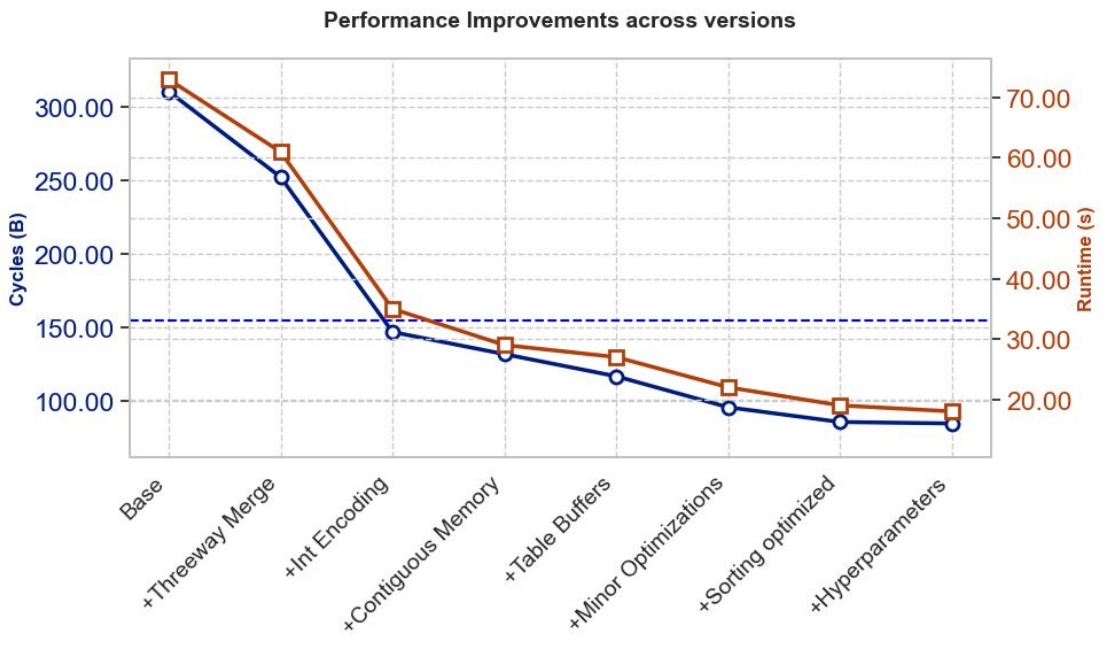
- Multithreading introduced race conditions and debugging challenges.
- Added significant code complexity, making the implementation harder to maintain.

I/O BOTTLENECKS

- Sorting relies heavily on file I/O.
- File I/O is inherently sequential, limiting the gains from multithreading.

PERFORMANCE (CYCLES)

Reference Implementation	Base Implementation	Final Implementation
155,414,886,890 cycles	310,116,076,286 cycles	82,717,600,559 cycles



CONCLUSIONS

WHAT WORKED WELL

+ Integer Encoding of strings
+ Precomputation of Int Encoding
+ Sorting Algorithm optimizations
+ Minor local Optimizations
- Multithreading
- Branchless Programming

THANK YOU FOR LISTENING :)

ANY QUESTIONS?