# Exploring D via Benchmarking of eBay's TSV Utilities

Jon Degenhardt, eBay Inc.
DConf 2018

# Today’s talk

- Describe performance evaluations of D conducted using eBay's TSV Utilities. Studies were part of a larger look at the D programming language

- Talk will describe the larger context behind these studies, what was learned, etc.

**About myself**

- Senior member of eBay's Search Science team. Recall, ranking, and search engine architecture.

- Over a decade in search at Yahoo and eBay

- Experienced in both engineering and data science

# About eBay's TSV Utilities

- Command line tools manipulating large, tabular data files. e.g. Machine learning data sets. Large but still manageable on a single machine. eg. 10GB, 100 million rows.
- Filtering, statistics, sampling, joins, etc. Used alongside standard Unix CLI tools (e.g. grep, sed, sort) as well as R, Pandas, etc.
- Written in D to explore the language. Niche tools, but real tools doing real work.
- Eleven tools (executables); 4400 lines of code (CodeCov count)
- Imperative and functional programming throughout; Templates; Object oriented w/ inheritance (tsv-summarize), Finite state machine (csv2tsv)
- Standard library use: Algorithms (many); Ranges; Containers (AA, Appender, Binary Heap, DList); Regular Expressions; Random numbers; Unicode; Hashing; Math routines; CLI Options (getopt)

# TSV Utilities: Key Tools

- tsv-filter - Filter rows by numeric and string comparisons
- tsv-summarize - Summary statistics on fields (sum, median, etc) with optional group-by
- tsv-select - Like Unix cut, but with ability to reorder fields
- tsv-sample - Uniform and weighted random sampling; Distinct sampling
- tsv-join - Join across multiple files
- tsv-uniq - Like Unix uniq, but with fields as key, no sorting required
- tsv-pretty - Print with aligned columns
- csv2tsv - Convert CSV format to TSV (remove CSV escapes, etc)

# Why D?

- Looking for simpler alternatives to C++ for performance sensitive components of eBay's search stack
- C++ enables great performance, but is complex to write and maintain
- Undesirable barrier for data scientists, who spend most of their time writing in other languages (Python, R, Scala, etc), making occasional contributions to C++ components
- D has an intriguing mix of high performance, simplified coding, and pragmatic features

# Evaluation approach

- Write code in a style achievable by a team of programmers
- Straightforward code, use the standard library (Phobos) where possible
- Avoid writing low level code (buffer management, memory management, etc)
- Identify and avoid inefficient code constructs (e.g. auto-decoding)
- Avoid unnecessary GC allocation, but don't avoid GC
- Evaluation considerations: Language and library facilities; ease of use; maintainability, testability, tooling, interoperability, performance, etc.

# March 2017 Comparative Benchmarks

- Idea: Assess D's performance by comparing against similar tools written in other native compiled languages
- Target “normal” code, not highly optimized micro-benchmarks
- Published benchmarks have at least two tools with similar functionality
- Benchmarks were run only after finalizing the TSV Utilities
- 6 benchmarks; 9 other tools in 3 languages (C, Go, Rust); 33 total benchmark times
- Caveats: Of course! But nothing to invalidate the big picture results.

# Result: TSV Utilities were the fastest on each test

Quite unexpected. On most tests it was not close.

## March 2017 Comparative Benchmarks: Top-4 in each test

| Benchmark | Tool/Time | Tool/Time | Tool/Time | Tool/Time |
|---|---|---|---|---|
| **Numeric row filter** | ***tsv-filter*** | GNU awk | mawk | Toolkit 1 |
| **(4.8 GB, 7M lines)** | 4.34 | 11.71 | 22.02 | 53.11 |
| **Regex row filter** | ***tsv-filter*** | GNU awk | mawk | Toolkit 1 |
| **2.7 GB, 14M lines** | 7.11 | 15.41 | 16.58 | 28.59 |
| **Column selection** | ***tsv-select*** | mawk | GNU cut | Toolkit 1 |
| **4.8 GB, 7M lines** | 4.09 | 9.38 | 12.27 | 19.12 |
| **Summary statistics** | ***tsv-summarize*** | Toolkit 1 | Toolkit 2 | Toolkit 3 |
| **(4.8 GB, 7M lines)** | 15.83 | 40.27 | 48.10 | 62.97 |
| **Join two files** | ***tsv-join*** | Toolkit 1 | Toolkit 2 | Toolkit 3 |
| **4.8 GB, 7M lines** | 20.78 | 104.06 | 194.80 | 266.42 |
| **CSV-to-TSV** | ***csv2tsv*** | csvtk | xsv | |
| **(2.7 GB, 14M lines)** | 27.41 | 36.26 | 40.40 | |

*Macbook Pro, 16 GB RAM, SSD drives. Times in seconds.*

# Take-aways

- Using a native compiled language won't by itself make a program fast
- TSV processing is a useful benchmark basis because there are multiple alternative tools
- D shined on these benchmarks:
  - Solid optimization throughout the standard library
  - LDC compiler optimizations are quite good
- D's programming paradigms are good fit for this class of problem:
  - Easy to write code for these tools
  - Ranges, lazy algorithms, etc.
  - Throughput oriented (GC pauses are immaterial)
- Meaningful stuff not tested: Concurrency/threading; Latency oriented applications (e.g. service requests); Larger variety of memory allocation patterns (e.g. many small objects)

# Link Time Optimization and Profile Guided Optimization

Fall 2017 evaluations

# LTO and PGO evaluation: Fall 2017

- LTO and PGO are LLVM technologies exposed by LDC (LLVM D Compiler)
- Link Time Optimization (LTO): Whole program optimization at link-time
- Profile Guided Optimization (PGO): Compiler optimizations using profile data from an instrumented build
- Recent LDC innovation: Support both LTO and PGO across application code and druntime/phobos libraries
- TSV Utilities are already fast - How much headroom remains?

# More about LTO and PGO

**Link Time Optimization**

- Inter-procedural optimizations difficult or impossible when considering only part of a program (e.g. an individual source file)
- LLVM approach: Compiler writes LLVM IR bitcode to .o files. Linker reads IR bitcode from all files to perform whole program optimization
- Common optimizations: Cross-module inlining, dead code elimination (smaller binaries)
- Full vs Thin LTO: Full LTO reads entire IR code, Thin LTO reads module “summaries”. Thin is faster, uses less memory, but has less information. Thin LTO generally optimizes nearly as well as Full LTO.

**Profile Guided Optimization**

- Main challenge is creating representative workloads
- Common optimizations: Improved inlining decisions; branch prediction

# LTO and PGO performance

Same tests as the 2017 Comparative Benchmarks

| LTO/PGO | tsv-summarize | csv2tsv | tsv-filter (numeric) | tsv-filter (regex) | tsv-select | tsv-join |
|---|---|---|---|---|---|---|
| **None** | 21.79 | 25.43 | 4.98 | 7.71 | 4.23 | 21.33 |
| **ThinLTO: App Only** | 22.40 | 25.58 | 5.12 | 7.59 | 4.17 | 21.24 |
| **ThinLTO: App+Libs** | 10.41 | 21.41 | 3.71 | 7.04 | 4.05 | 20.11 |
| **ThinLTO+PGO: App+Libs** | 9.25 | 14.32 | 3.50 | 7.09 | 3.97 | Not tested |
| **Improvement** | | | | | | |
| **ThinLTO: App+Libs** | 52% | 16% | 26% | 9% | 4% | 6% |
| **PGO vs ThinLTO** | 11% | 33% | 6% | -1% | 2% | Not tested |
| **ThinLTO+PGO** | 58% | 44% | 30% | 8% | 6% | Not tested |

- *Macbook Pro, 16 GB RAM, SSD drives; LDC 1.5.0. Times in seconds.*
- *Linux benchmarks showed similar improvements*

# LTO Executable sizes

## macOS sizes (bytes)

| LTO | tsv-summarize | csv2tsv | tsv-filter | tsv-select | tsv-join |
|---|---|---|---|---|---|
| None | 7,988,448 | 6,709,936 | 8,137,804 | 6,890,192 | 6,945,336 |
| ThinLTO: App Only | 6,949,712 | 6,643,344 | 6,639,844 | 6,676,000 | 6,688,392 |
| ThinLTO: App+Libs | 3,082,068 | 2,679,184 | 3,172,648 | 2,734,356 | 2,738,700 |
| Reduction | 61% | 60% | 61% | 60% | 61% |

## Linux sizes (bytes)

| LTO | tsv-summarize | csv2tsv | tsv-filter | tsv-select | tsv-join |
|---|---|---|---|---|---|
| None | 1,400,672 | 995.760 | 1,743,288 | 1,026,344 | 1,049,176 |
| Full-LTO: App Only | 1,300,792 | 998,432 | 1,547,656 | 1,024,312 | 1,036,648 |
| Full-LTO: App+Libs | 1,154,808 | 826,064 | 1,359,554 | 856,064 | 868,736 |
| Reduction | 18% | 17% | 22% | 17% | 17% |

# Take-aways

- Substantial performance gains from both LTO and PGO
- Key was using LTO and PGO on both application code and libraries
- Hard to predict gains in advance. Some apps gained more from LTO, others from PGO.
- LTO compile times not significantly longer for TSV Utilities apps. Larger apps are likely to be more impacted.
- A couple edge-case bugs were encountered with LTO. Having a good test suite was helpful to identify these.

# Comparative Benchmark Update

*Update to the March 2017 study*

# Comparative benchmark update (April 2018)

- Same benchmarks as before, plus a narrow file column selection test
- Used the fastest programs from the March 2017 benchmarks
- Both MacOS and Linux (2017 tests were MacOS only)
- Performance related changes in TSV Utilities:
  - Compiler/language: LDC 1.1 -> 1.5; druntime/phobos 2.071 -> 2.075
  - LTO & PGO (including druntime/phobos)
  - Output buffering - Addresses several output inefficiencies (stdio.write[f|ln], algorithm.joiner)

# Result: TSV Utilities have gotten faster

## TSV Utilities performance improvements

| | MacOS | | | Linux | | |
|---|---|---|---|---|---|---|
| | March 2017 (v1.1.11) | April 2018 (v1.1.19) | Delta | March 2017 (v1.1.11) | April 2018 (v1.1.19) | Delta |
| **Numeric row filter** | 4.25 | 3.35 | 21% | 6.32 | 5.48 | 13% |
| **Regex row filter** | 10.11 | 8.28 | 18% | 9.72 | 8.80 | 9% |
| **Column selection** | 4.26 | 2.93 | 31% | 5.52 | 4.79 | 13% |
| **Column selection: narrow** | 25.12 | 10.18 | 59% | 15.41 | 8.26 | 46% |
| **Summary statistics** | 16.97 | 9.82 | 42% | 22.68 | 15.78 | 30% |
| **Join two files** | 23.94 | 21.17 | 12% | 28.77 | 26.68 | 7% |
| **CSV-to-TSV** | 30.70 | 10.91 | 64% | 34.65 | 20.30 | 41% |

*Times in seconds*
*MacOS: Mac mini, 16 GB RAM, SSD drives, 3 GHz Intel i7 (2 cores)*
*Linux: Commodity cloud machine: 16 CPUs (Intel Haswell 2095 MHz), 32 GB RAM*
*TSV Utilities v1.1.11 used to generate the March 2017 benchmark equivalent*

# Result: TSV Utilities are still the fastest overall
# But… the other tools have gotten faster too

## MacOS: Top-4 in each test

| Benchmark | Tool/Time | Tool/Time | Tool/Time | Tool/Time |
|---|---|---|---|---|
| **Numeric row filter** | ***tsv-filter*** | mawk | GNU awk | csvtk |
| **(4.8 GB, 7M lines)** | 3.35 | 15.06 | 24.25 | 39.10 |
| **Regex row filter** | xsv | ***tsv-filter*** | GNU awk | mawk |
| **(2.7 GB, 14M lines)** | 7.03 | 8.28 | 16.47 | 19.40 |
| **Column selection** | ***tsv-select*** | xsv | csvtk | mawk |
| **(4.8 GB, 7M lines)** | 2.93 | 7.67 | 11.00 | 12.37 |
| **Column selection: narrow** | xsv | ***tsv-select*** | GNU cut | csvtk |
| **(1.7 GB, 86M lines)** | 9.22 | 10.18 | 10.65 | 23.01 |
| **Summary statistics** | ***tsv-summarize*** | xsv | csvtk | GNU datamash |
| **(4.8 GB, 7M lines)** | 9.82 | 35.32 | 45.59 | 71.60 |
| **Join two files** | ***tsv-join*** | xsv | csvtk | |
| **(4.8 GB, 7M lines)** | 21.78 | 60.03 | 82.43 | |
| **CSV-to-TSV** | ***csv2tsv*** | xsv | csvtk | |
| **(2.7 GB, 14M lines)** | 10.91 | 14.38 | 32.49 | |

*Times in seconds*
*Mac mini, 16 GB RAM, SSD drives, 3 GHz Intel i7 (2 cores)*

# Linux results - Similar

## Linux: Top-4 in each test

| Benchmark | Tool/Time | Tool/Time | Tool/Time | Tool/Time |
|---|---|---|---|---|
| **Numeric row filter** | ***tsv-filter*** | mawk | GNU awk | csvtk |
| **(4.8 GB, 7M lines)** | 5.48 | 11.31 | 42.80 | 53.36 |
| **Regex row filter** | xsv | ***tsv-filter*** | mawk | GNU awk |
| **(2.7 GB, 14M lines)** | 7.97 | 8.80 | 17.74 | 29.02 |
| **Column selection** | ***tsv-select*** | mawk | xsv | GNU cut |
| **(4.8 GB, 7M lines)** | 4.79 | 9.51 | 9.74 | 14.46 |
| **Column selection: narrow** | GNU cut | ***tsv-select*** | xsv | mawk |
| **(1.7 GB, 86M lines)** | 5.60 | 8.26 | 13.60 | 23.88 |
| **Summary statistics** | ***tsv-summarize*** | xsv | GNU datamash | csvtk |
| **(4.8 GB, 7M lines)** | 15.78 | 44.38 | 48.51 | 59.71 |
| **Join two files** | ***tsv-join*** | xsv | csvtk | |
| **(4.8 GB, 7M lines)** | 26.68 | 68.02 | 98.51 | |
| **CSV-to-TSV** | ***csv2tsv*** | xsv | csvtk | |
| **(2.7 GB, 14M lines)** | 20.30 | 26.82 | 44.82 | |

*Times in seconds*
*Commodity cloud machine: 16 CPUs (Intel Haswell 2095 MHz), 32 GB RAM*

# Concluding thoughts

- D makes it easy to develop fast programs!
- LTO and PGO really work!
- Developing real tools, though simple, has proven quite valuable for exploring D’s many features
- Having benchmark data, even if incomplete or flawed, is preferable to having no data or only anecdotal data.

# References

- eBay's TSV Utilities GitHub repository - Tools, documentation, and benchmark details
- Link Time Optimization (LTO), C++/D cross-language optimization - Johan Engelen’s blog post describing LTO in the LDC compiler
- ThinLTO: Scalable and Incremental Link-Time Optimization. CppCon 2017, Teresa Johnson. The talk to see if you want to understand LTO.