Improving Performance in TOPCAT & STILTS

University of BRISTOL

Mark Taylor m.b.taylor@bristol.ac.uk

Introduction

TOPCAT[1], STILTS and STIL are Java software for working with tabular data such as source catalogues, and aim to provide astronomers with the capability to manipulate datasets as large as possible on modest hardware such as standard desktop or laptop machines. How large that is depends on various factors including hardware resources, data format and the patience of the user. However, as a guideline to current capabilities:

- the interactive GUI tool TOPCAT can comfortably be used for exploratory analysis of tables with tens of millions of rows and hundreds of columns;
- the command-line tool STILTS and the underlying I/O library STIL can be used with arbitrary-sized tables; for instance STILTS can generate a HEALPix weighted density map from a FITS file containing the 1.8 billion row Gaia source catalogue in $\sim$5 mins.

Development of the software began around 2 decades ago, and most of the algorithms were originally single-threaded. Over that time multi-core processors have become ubiquitous, and in the last 3–4 years effort has been expended on trying to exploit this by multi-threading the code to run computationally-intensive algorithms concurrently, alongside other interventions to improve efficiency. The changes reported here cover the period between TOPCAT v4.6-3 (May 2019) and TOPCAT v4.8-7 (Oct 2022).

The two main operations that are computationally intensive, that is whose speed impacts the user experience, are visualisation and crossmatching.

Multithreading: Java Language Support

The Java platform provides fairly good language support for multi-threading. Java 8 (2014) introduced streams which look like a good abstraction for multi-threaded processing of large collections of similar items such as table rows. But these turned out to be hard to work with in STIL because of inflexibility of the library interface and opaqueness of the standard implementation; for instance sequential evaluation is sometimes forced in circumstances that are hard to predict, stream elements can be retained in a way that inhibits object re-use, and aborting stream execution is difficult. However by following some of the *streams* patterns and building on the lower-level classes on which they are based (Java 7 `ForkJoinPool`) a bespoke stream-like framework was implemented to support concurrent execution of CPU-intensive algorithms.

Multithreading Visualisation

The interactive visualisation in TOPCAT (usually) operates in two stages, which may be designated caching and rendering. First the coordinate data is read from the input table, possibly pre-processed, and cached to a simplified memory-based structure. Then, graphical frames are repeatedly rendered as required from this cached data in response to user navigation actions (pan and zoom). For most plot types the rendering stage is suitable for multithreading, and since no I/O is involved this works well, delivering near optimal acceleration at least in some regimes (e.g. $\approx 8\times$ on 10 cores for a 100Mrow weighted density plot). The caching stage can be effectively parallelised in *some* cases, but problems arise if for instance an unpredictable selection of the rows in the full dataset is being plotted, so by default caching is still done sequentially.

Multithreading therefore boosts the size of dataset that can be comfortably visualised interactively by an order of magnitude on a typical multi-core machine, though initiating a plot may still take some time (perhaps a few tens of seconds for a 100 Mrow plot).

Multithreading Crossmatching

Unlike visualisation rendering (see above), the cross-matching implementation works directly with instances of the `StarTable` abstraction used throughout STIL to represent tabular data. Multi-threading the relevant code therefore required changes to the `StarTable` interface to prepare it for concurrent access, which was a major job. Moreover, the cross-matching algorithms involve multiple steps, not all of which are amenable to parallelisation, so overall speedup is harder to achieve in accordance with Amdahl's Law[2].

Once `StarTable` had been thus modified and those loops that were suitable had been parallelised, some acceleration did result, but overall speedup was a bit disappointing. Profiling identified bottlenecks, leading to a number of other changes to improve performance. In its current state, a parallelism of about six speeds up end-to-end elapsed time for a typical crossmatch by around a factor of two, though this is highly dependent on the details of the match. Higher parallelisms yield diminishing returns for elapsed time, so default behaviour is now to enable parallel execution with ≤ 6 threads.

The main reason for this limited scalability is that some parts of the algorithm cannot easily be parallelised. Other factors resulting from fragmenting the processing also play a part: more combination operations and more garbage collection are required, and access to memory and disk is more scattered.

[1] http://www.starlink.ac.uk/topcat/
[2] https://en.wikipedia.org/wiki/Amdahl's_law
[3] https://github.com/cds-astro/cds-healpix-java
[4] https://github.com/jvm-profiling-tools/async-profiler

StarTable Parallelisation

The `StarTable` interface serves as the basic data access abstraction in STIL; a `StarTable` may represent any source of table data, such as a file on disk or a stream of processing operations derived from a stored or virtual row sequence.

STIL v4.0 (Jan 2021) introduced a number of changes required to support multithreaded operations on `StarTable`s including new methods:

- `getRowSplittable()`: returns an object that manages splitting the row sequence into blocks to enable multi-threaded sequential access to table data
- `getRowAccess()`: returns a thread-safe object providing random access to table data

Since there are hundreds of StarTable subclasses in the library, careful design and implementation of these methods was required. Many other classes also needed to be identified and adjusted for concurrent use.

Other Crossmatch Improvements

Profiling the crossmatching code identified various bottlenecks which were addressed separately to the parallelisation. These included:

- Replace HEALPix library with one that is more efficient, better supported and more threadsafe (PixTools $\rightarrow$ CDS HEALPix library[3])
- Use a `HashMap` instead of a `TreeMap` for one data structure
- Replace the arcsine implementation (`java.lang.Math` $\rightarrow$ Apache `FastMath`)
- Enhance pre-processing coverage assessment to understand spherical geometry

Together these delivered a factor of around $3\times$ speedup for typical matches — greater than the effect of the parallelisation.

I/O

Careful attention to behaviour of various I/O classes, especially with respect to buffering, delivered significant speedups, e.g. writing to FITS and reading BINARY-encoded VOTables were improved by a factor of 3–4$\times$. In some cases this was achieved by replacing use of the standard library classes, which are not always performant for high-volume usage, with suitable custom alternatives.

Much of the most resource-intensive work required by these applications is I/O-bound. In this regime multi-threading doesn't help much and may well degrade performance by fragmenting data access and impeding efficient use of system caches. This can be a major effect when reading from spinning disks for which seek operations are expensive, though less so when reading from SSDs or a hot cache. These considerations make it challenging to set up reliably beneficial default threading configurations, e.g. number of concurrent threads. STILTS therefore provides options for users to experiment with concurrency.

Profiling

To identify bottlenecks a good runtime profiler is essential. Historically most (all?) java CPU profiling tools have suffered from the *Safepoint Bias Problem* which often gives rise to misleading information about where CPU cycles are used. Fortunately the async-profiler[4] application which avoids this problem has become available in the last few years. Even so, understanding the contributions of object allocation, garbage collection, I/O and memory access to overall run times is still challenging.

Miscellaneous Tips & Insights

- Having a good profiler helps (but don't trust it too much)
- Retrofitting concurrency to sequential code is hard
- It's hard to predict what's going to make a big difference
- Identifying bottlenecks can be as time-consuming as eliminating them
- Parallelisation can easily make things slower instead of faster
- Don't assume that Java standard library classes are fast
- High CPU usage doesn't necessarily mean elapsed time speedup
- Remember Amdahl's Law
- Performance optimisation is a black experimental art

Conclusions

Parallelising interactive visualisation has worked well; parallelising crossmatching has taken more effort and been less successful. As well as the parallelisation, a number of other performance interventions have been able to improve the user experience when working with very large datasets in TOPCAT and STILTS. Over the past few years, most of the resource-intensive operations have been accelerated by a factor of several; typically $6\times$ for crossmatching and $\sim N\times$ for interactive visualisation on an N-core machine.