An Introduction to Fast JSON Parsing

Achutha Balaji, Angelo Kastroulis
{abalaji,akastroulis}@carrera.io
Carrera Group

ABSTRACT

This paper introduces simdjson a high-performance novel approach to parsing JSON and compares it with a common method of JSON parsing in Java-based applications (Jackson). Further, the particular case tested is that of mapping (which involves parsing, but involves traversal of nodes).

1 INTRODUCTION

JSON and XML have become ubiqitous in data systems and mechanism for describing and respresenting data. Compared with XML, JSON is a light-weight key-value style data exchanging format. The efficiency of mapping data between different data models is key point to improving the performance of web service applications [4]. Mapping between data models becomes even more critical as applications become decoupled to allow fine-grained control over scalability of components. Since each component retains control over its own data representation, such a design lends itself to mapping the same data between models over and over again. The performance cost, then, multiplied many times.

2 BACKGROUND AND RELATED WORK

Solving this problem is not easy. Consider the journey of data through the system in Figure 1. Each step in the process copies the data into another portion of memory and reads it in its entirety. Eliminating steps, therefore would improve performance.

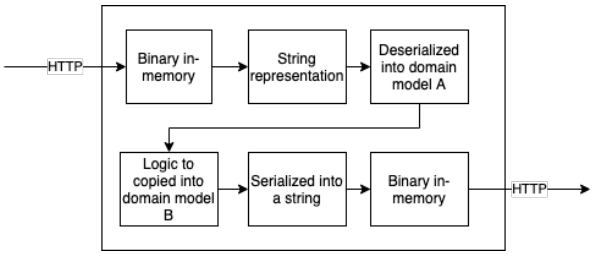

Figure 1: Data Copies in Mapping Model A to B

2.1 Java-based Generic Approaches

Jackson has been known as "the Java JSON library" or "the best JSON parser for Java". Or simply as "JSON for Java". [1]. It is a suite of tools, and a framework for parsing JSON for Java. Jackson also has capability built-in for Apache Avro and other schemas. It is a well-built framework designed for high-performance workloads.

2.2 simdjson Approach

A relatively newer arrival to the landscape is simdjson, which claims to be even faster than RapidJSON, a high-performance framework in C++ [3]. RapidJSON implements a zero-copy approach where the data is parsed in-place (no copy is made, but pointers to the positions are stored on a parse pass) [2]. Simdjson extends that principle to include SIMD (Single Instruction Multiple Data) chipset operations. SIMD has found success in the internals of data systems algorithms [5], but has not been applied to JSON parsing until simdjson [3].

3 METHODOLOGY/DESIGN

3.1 Loading and Mapping JSON Files

JSON parsing libraries such as Jackson and simdjson provide tools to both load and map JSON files to other JSON files, Avro files, etc. An example of such an application is to read an incoming JSON file/object like this:

Listing 1: "books.json"

```
{
  "store": {
    "book": [
      {
        "category": "reference",
        "author": {"name": "N. Rees"},
        "title": "Sayings of the Century",
        "price": 8.95
      },
      {
        "category": "fiction",
        "author": {"name": "E. Waugh"},
        "title": "Sword of Honour",
        "price": 12.99
      },
      {
        "category": "fiction",
        "author": {"name": "H. Melville"},
        "title": "Moby Dick",
        "isbn": "0-553-21311-3",
        "price": 8.99
      },
      {
        "category": "fiction",
        "author": {"name": "J. Tolkien"},
```


, ,

```
        "title": "The Lord of the Rings",
        "isbn": "0-395-19395-8",
        "price": 22.99
      }
    ],
    "bicycle": {
      "color": "red",
      "price": 19.95
    }
  },
  "expensive": 10
}
```

And map it to a destination JSON file/object like this:

Listing 2: "output.json"

```
{
  "publication": [
    {
      "category": "reference",
      "author": "N. Rees",
      "price": 8.95
    },
    {
      "category": "fiction",
      "author": "E. Waugh",
      "price": 12.99
    },
    {
      "category": "fiction",
      "author": "H. Melville",
      "price": 8.99
    },
    {
      "category": "fiction",
      "author": "J. Tolkien",
      "price": 22.99
    }
  ]
}
```

Loading and mapping JSON objects is fairly straightforward with both Jackson and simdjson. In C++ (simdjson), loading and mapping looks like this:

Listing 3: Loading JSON with C++ (simdjson)

```
ondemand::parser parser;
auto json;
json = padded_string::load("books.json");
ondemand::document doc;
doc = parser.iterate(json);
/* map JSON object */
```

While Java (Jackson) looks somewhat similar:

Listing 4: Loading JSON with Java (Jackson)

```
ObjectMapper objectMapper;
objectMapper = new ObjectMapper();
JsonNode jsonNode;
jsonNode = objectMapper.readTree(new
File("books.json"));
/* map JSON object */
```

Both blocks of code are wrapped in a timer: C++ (simdjson) uses the *chrono::high_resolution_clock::now()* and Java (Jackson) uses *System.nanoTime()* to measure the time elapsed for loading and mapping the JSON object.

Describe the mapping here...

4 EVALUATION

4.1 Test Setup

We ran 2 tests: one with the Java code (Jackson) 10 times each for 4 differently-sized JSON files and one with the C++ code (simdjson) 10 times each for the same 4 differently-sized JSON files. For both tests, we only measured the time elapsed for loading the JSON file and mapping its contents to another JSON file. This ensures that our tests comparing the two libraries are as fair as possible since we are purposely factoring out other events that could affect the time elapsed (e.g. JVM startup, etc.). The 4 JSON files are classified as small (1 kilobyte), medium (63 kilobytes), large (6.3 megabytes), and very large (63.2 megabytes) each with arrays of size 4, 400, 40,000, and 400,000 elements, respectively.

4.2 Test Environment

Add machine specs here...

4.3 Results

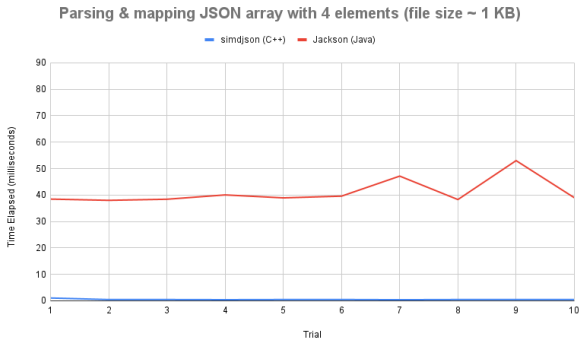

Figure 2: Loading and mapping a small JSON file.

Here are a few charts of the results (Figures 2-6). Figures 2, 3, 4, and 5 display the time elapsed for loading and mapping small, medium, large, and very large JSON files, respectively, over 10 trials for both the Java code (Jackson) and the C++ code (simdjson). Figure 6 displays the average time elapsed over loading and mapping each file for both Jackson and simdjson. As observed in the charts, simdjson clearly dominates Jackson in loading and mapping small and medium files. However, simdjson and Jackson take almost equal time to load and map a large file, with Jackson beating simdjson in every trial for the very large file. The time elapsed for loading

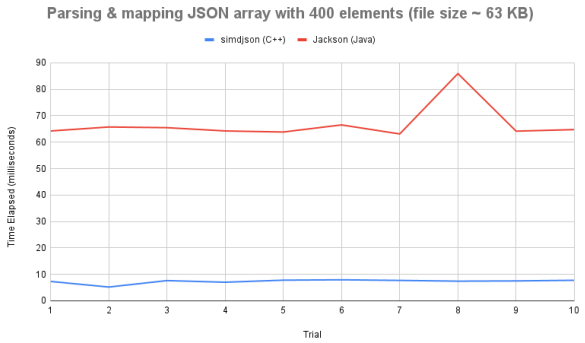

Figure 3: Loading and mapping a medium JSON file.

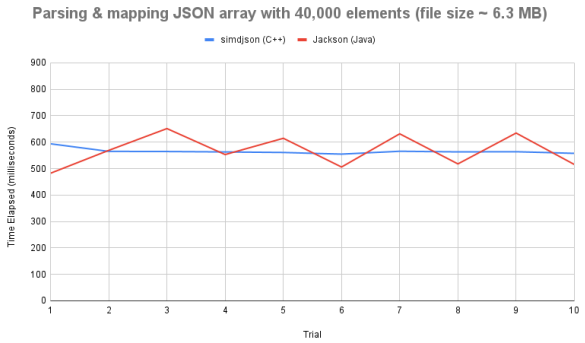

Figure 4: Loading and mapping a large JSON file.

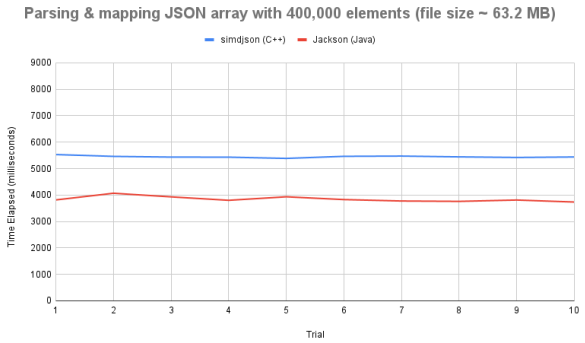

Figure 5: Loading and mapping a very large JSON file.

and mapping files grows much faster with simdjson as the file size increases than with Jackson. Going from the small file to the medium file, Jackson loads and maps the file about 1.6 times as slow (even with a file size over 60 times as massive) while simdjson does the same operations about 14.4 times as slow. And this effect is even more exaggerated when we move to larger file sizes, with Jackson overtaking simdjson in performance at file sizes around 6 megabytes. It is also important to note that Jackson experiences noticeable fluctuations in performance while simdjson performs in a relatively stable manner across trials. However, both libraries do not appear to experience drastic fluctuations in performance for very large files.

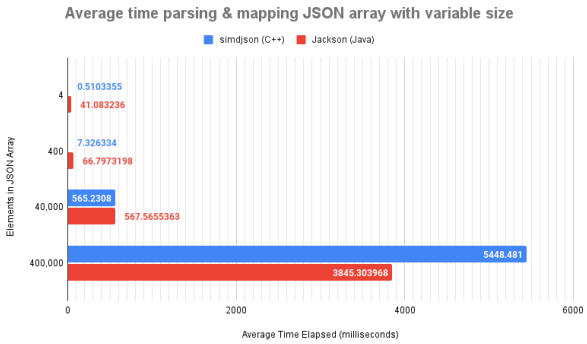

Figure 6: Average time loading and mapping JSON files.

5 CONCLUSION

On smaller to medium-sized JSON files (less than a megabyte), simdjson was far superior to Jackson. However, this advantage completely disappeared for larger JSON files (greater than a few megabytes). This could possibly be attributed to differences in Java and C++ that were unaccounted for during our experiments. These differences could significantly increase performance costs as incoming file sizes grow larger. We are still very far from parsing "gigabytes per second" with simdjson as claimed (citation needed).

5.1 Subsequent Work

An immediate next step is to continue attempting to replicate the results produced by simdjson. A possible further step is to create a binding of simdjson for Java using *Java Native Interface (JNI)*, or another tool, and comparing its performance with Jackson and other Java libraries for parsing JSON.

Another step that can be taken is to provide a reusable and extensible mapping mechanism (either a library or format).

Another option is to build a Java-native version (using the new Vector API).

REFERENCES

[1] Jackson Project Home @github. https://github.com/FasterXML/jackson. (????). Accessed: 2021-08-27.

[2] RapidJSON Features. https://rapidjson.org/md_doc_features.html. (????). Accessed: 2021-08-27.

[3] Geoff Langdale and Daniel Lemire. 2019. Parsing gigabytes of JSON per second. *The VLDB Journal* 28, 6 (2019), 941–960.

[4] Dunlu Peng, Lidong Cao, and Wenjie Xu. 2011. Using JSON for data exchanging in web service applications. *Journal of Computational Information Systems* 7, 16 (2011), 5883–5890.

[5] Jingren Zhou and Kenneth A Ross. 2002. Implementing database operations using SIMD instructions. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 145–156.

A APPENDIX