

# COM301T: Operating Systems

## A Multithreaded approach to parsing and converting PGN files to JSON

Aditya DS (ESD18I006)  
Vignesh Nagarajan (EDM18B055)

December 6, 2020

<b>Instructor:</b>	Dr. Sivaselvan B.
<b>Teaching Assistants:</b>	Santosh Kumar
	Mercy Faustina

# Contents

<b>0</b>	<b>Introduction</b>	<b>3</b>
0.1	What are PGN files? . . . . .	3
0.1.1	Tag Pairs . . . . .	3
0.1.2	Movetext . . . . .	3
0.1.3	A sample PGN file . . . . .	4
0.2	What are JSON files? . . . . .	5
0.2.1	A sample JSON file . . . . .	6
0.3	Aim . . . . .	7
<b>1</b>	<b>Building the PGN Parser</b>	<b>7</b>
1.1	Demonstration in Browser Environment . . . . .	8
1.2	Demonstration in Node.js Environment . . . . .	9
1.2.1	Driver Code . . . . .	9
1.2.2	Output . . . . .	9
<b>2</b>	<b>Parsing large PGN files that contain multiple games</b>	<b>10</b>
2.1	Single-threaded Approach . . . . .	10
2.2	Multi-threaded Approach . . . . .	10
2.3	Demonstration . . . . .	11
<b>3</b>	<b>Benchmarking</b>	<b>12</b>
<b>4</b>	<b>Results</b>	<b>13</b>
<b>5</b>	<b>Conclusion</b>	<b>15</b>
<b>6</b>	<b>Important Links</b>	<b>16</b>
6.1	Parser . . . . .	16
6.2	Project . . . . .	16

## List of Figures

0	Object . . . . .	5
1	Array . . . . .	5
2	Values . . . . .	6
3	Demo at: <a href="https://aditya-ds-1806.github.io/Chess-PGN-Parser/">https://aditya-ds-1806.github.io/Chess-PGN-Parser/</a> . . . . .	8
4	A portion of the terminal output that shows the moves . . . . .	9
5	A portion of the terminal output that shows the annotations and NAGs . . . . .	10
6	Output for roughly 13000 games . . . . .	11
7	A glimpse of the output after parsing 13000 games. On the left is the output for single-threaded version and on the right is the output for the multi-threaded version. . . . .	12
8	Plot of games parsed vs the average time taken . . . . .	15

## List of Tables

0	Some common Tag pairs . . . . .	3
1	Time taken to fully parse 827 games and output JSON . . . . .	13
2	Time taken to fully parse 1654 games and output JSON . . . . .	13
3	Time taken to fully parse 3308 games and output JSON . . . . .	13
4	Time taken to fully parse 6616 games and output JSON . . . . .	13
5	Time taken to fully parse 13232 games and output JSON . . . . .	14
6	Time taken to fully parse 26464 games and output JSON . . . . .	14
7	Time taken to fully parse 52928 games and output JSON . . . . .	14
8	Time taken to fully parse 105856 games and output JSON . . . . .	14
9	Table summarising the average time taken to parse PGN files of varying sizes for single-threaded and multi-threaded approaches. . . . .	15

# 0 Introduction

## 0.1 What are PGN files?

Portable Game Notation (PGN) is a standard plain text format for recording chess games (both the moves and related data), which can be read by humans and is also supported by most chess software. PGN was devised around 1993, by Steven J. Edwards, and was first popularized and specified via the Usenet newsgroup rec.games.chess. PGN is *structured for easy reading and writing by human users and for easy parsing and generation by computer programs*. The chess moves themselves are given in algebraic chess notation. The usual filename extension is *.pgn*.

PGN text begins with a set of tag "pairs" (a tag name and its value), followed by the "movetext" (chess moves with optional commentary).

### 0.1.1 Tag Pairs

Tag pairs begin with an initial left bracket [, followed by the name of the tag in plain ASCII text. The tag value is enclosed in double-quotes, and the tag is then terminated with a closing right bracket ]. A quote inside a tag value is represented by the backslash immediately followed by a quote. A backslash inside a tag value is represented by two adjacent backslashes. There are no special control codes involving escape characters, or carriage returns and linefeeds to separate the fields, and superfluous embedded spaces are usually skipped when parsing. Some common Tag pairs are mentioned below:

Event	the name of the tournament or match event.
Site	the location of the event.
Date	the starting date of the game.
Round	the playing round ordinal of the game within the event.
White	the player of the white pieces.
Black	the player of the black pieces.
Result	the result of the game.

Table 0: Some common Tag pairs

### 0.1.2 Movetext

The movetext describes the actual moves of the game. This includes move number indicators (numbers followed by either one or three periods; one if the next move is White's move, three if the next move is Black's move) and movetext in *Standard Algebraic Notation* (SAN).

An annotator who wishes to suggest alternative moves to those actually played in the game may insert variations enclosed in parentheses. They may also comment on the game by inserting *Numeric Annotation Glyphs* (NAGs) into the movetext. Each NAG reflects a subjective impression of the move preceding the NAG or of the resultant position. Comments are inserted by either a ; (a comment that continues to the end of the line) or a { (which continues until a }). Comments do not nest.

### 0.1.3 A sample PGN file

```
[Event "Herceg Novi blitz"]
[Site "Herceg Novi blitz"]
[Date "1970.04.08"]
[EventDate "?"]
[Round "4.2"]
[Result "1-0"]
[White "Robert James Fischer"]
[Black "Samuel Reshevsky"]
[ECO "B40"]
[WhiteElo "?"]
[BlackElo "?"]
[PlyCount "119"]
```

```
1. e4 {Notes by Bobby Fischer} 1... c5 2. Nf3 e6 3. c4 Nc6 4. Nc3 Nf6 5. g3 g6
{5...d5 equalizes} 6. Bg2 Bg7 7. 0-0 0-0 8. d3 {if 8 d4? cxd4 9 Nxd4 Nxe4! wins
a Pawn.} 8... d6 9. h3 e5 {Weaker is 9...b6? 10d4 with an advantage for White
as in the game Smyslov-Reshevsky, the match of the century, Belgrade 1970. Now
the position is symmetrical with White two tempi ahead.} 10. a3 a5 11. Rb1 Bd7
12. Bd2 Ne8 13. Nd5 Ne7 14. b4 Nxd5 15. cxd5 $4 {Very anti-positional. Correct
of course was 15 exd5. Now Black gets a strong passed a-Pawn, also White's
b-Pawn is isolated.} 15... cxb4 16. axb4 a4 17. b5 $6 {Otherwise 17. . .
Nc7-b5. White sacrifices a Pawn for complications.} 17... Nc7 18. b6 Nb5 19.
Rb4 Qxb6 20. Qa1 a3 21. Rfb1 {If 20 Rxa4?,... Nc3 wins the exchange.} 21...
Rfc8 $1 22. Bf1 Rc2 23. d4 a2 24. R1b3 exd4 25. Bxb5 Bxb5 $2 {25. . . d3! won
by force. For example a) 26 Bd7 Ba1 27 Rxb6 Bb2 or . . . Bd4 etc. b) 26 e5 Rd2!
27 Nd2 Be5 28 Bd7 Ba1 29 Rb6 Bd4 etc.} 26. Rxb5 Qd8 27. Rd3 Qe7 28. Ne1 Qxe4
29. Rbb3 Rxd2 $1 30. Rxd2 d3 31. Rxa2 $5 {A last swindle try.} 31... Bxa1 $4
{31. . . Ra2! won easily 32.Qxa2 Qe1+ 33.Kg2 d2, But as Dr. Tarrasch said
"You must see it"} 32. Rxa8+ Kg7 33. Rxa1 d2 34. Ng2 Qxd5 35. Rbb1 b5 36. Rd1
b4 37. Ne3 Qd3 38. Nf1 b3 39. Rab1 b2 40. Rxb2 d5 41. Rdx2 Qc3 42. Ne3 d4 43.
Nd1 Qc4 44. Rb1 h5 45. h4 f5 46. Rdb2 f4 47. Rb3 fxg3 48. Rxg3 Qc2 49. Rb7+ Kh6
50. Nb2 Qe4 51. Rb6 Qe1+ 52. Kg2 Qe4+ 53. Kf1 Qb1+ 54. Ke2 Qc2+ 55. Ke1 Qe4+
56. Kd1 Qf5 57. Rbxg6+ Qxg6 58. Rxg6+ Kxg6 59. Ke2 Kf5 60. Kf3 1-0
```

## 0.2 What are JSON files?

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

JSON is built on two structures:

0. A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
1. An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

In JSON, they take on these forms:

0. An object is an unordered set of name/value pairs. An object begins with a { left brace and ends with } right brace. Each name is followed by : colon and the name/value pairs are separated by a , comma.

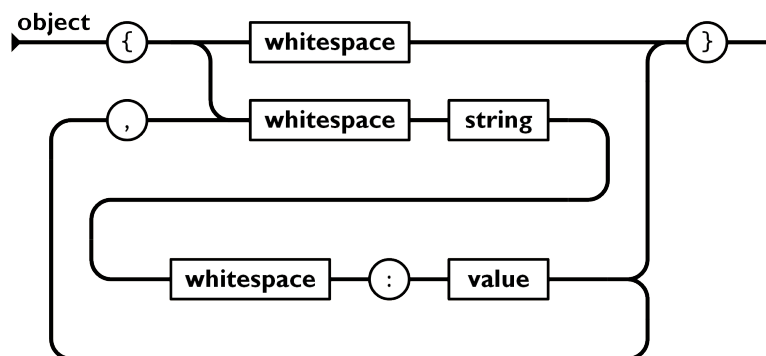


Figure 0: Object

1. An array is an ordered collection of values. An array begins with a [ left bracket and ends with ] right bracket. Values are separated by a , comma.

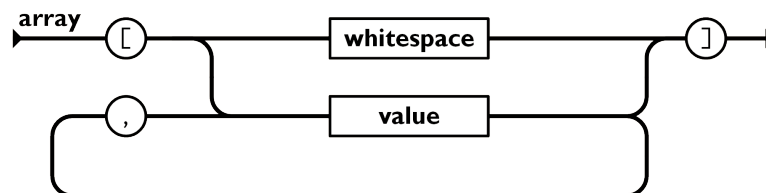


Figure 1: Array

2. A value can be a string in double quotes, or a number, or true or false or null, or an object or an array. These structures can be nested.

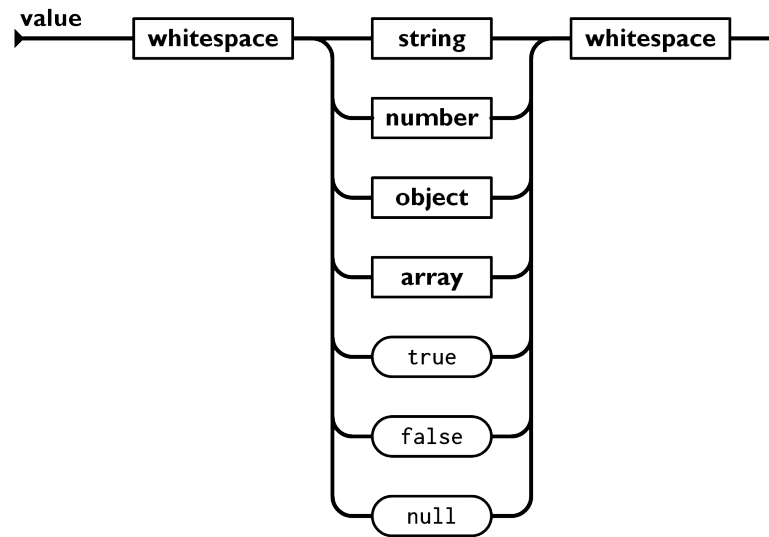


Figure 2: Values

### 0.2.1 A sample JSON file

```
{
  "quiz": {
    "maths": {
      "q1": {
        "question": "5 + 7 = ?",
        "options": [
          "10",
          "11",
          "12"
        ],
        "answer": "12"
      },
      "q2": {
        "question": "12 - 8 = ?",
        "options": [
          "2",
          "3",
          "4"
        ],
        "answer": "4"
      }
    }
  }
}
```

## 0.3 Aim

The aim of this project is:

1. To build a PGN file parser from scratch that converts PGN data to JSON, without relying on external libraries.
2. To use this parser to parse large PGN files containing thousands of games. Two approaches will be considered for parsing large files:
  - (a) Single threaded - Naive approach
  - (b) The Multi-threaded approach
3. To benchmark the two approaches to determine which is approach is more efficient.

Further, it was decided to build this application in *Node.js* using the *JavaScript* scripting language. *Git* and *GitHub* were used in the development process to ensure better collaboration.

## 1 Building the PGN Parser

```
export function pgn2json(pgnText) {
  var pgn = pgnText.split('\n');
  var game = {
    str: {},
    moves: [],
    annotations: [],
    nag: []
  }
  var moves, moveClone, movPos = [];

  pgn.forEach((_, i, pgn) => pgn[i] = pgn[i].trim().replace('\r', ''));
  pgn = removeEscapeMechanism(pgn);
  [game.str, moves] = getStrAndStringifyMoves(pgn);
  moveClone = moves.join('').split(' ');
  for (let i = 1; moveClone.includes(`${i}.`); i++) {
    movPos.push(moveClone.indexOf(`${i}.`));
  }

  game.nag = getNags(moveClone, movPos);
  game.annotations = getCommentary(moves, moveClone, movPos);

  moves = moves.join('').split(' ');
  moves = moves.filter(val => !val.includes('.') && !val.includes('$'));
  moves.pop();
  game.moves = moves;
  return JSON.stringify(game, null, 4);
}
```



Although it is not possible to list every piece of code, the heart of the application is the `pgn2json` method that is shown above. The function accepts a string that contains the entire PGN file and returns a prettified JSON string.

A few helper methods like `removeEscapeMechanism` and `getCommentary` are used which have been defined in another helper file. The code has been written in *ES6 syntax*. Node.js uses the *CJS (Common JS Module Loader)* which doesn't support ES6 syntax yet. We could have used a CJS Module bundler to transpile this code into a CJS module but then the code wouldn't work in browsers, since they use a different module loader called *AMD (Asynchronous Module definition)*.

Therefore to get around these problems, a module bundler called *rollup* was used to transpile the ES6 code into a UMD (Universal Module Definition) module. By doing so, the transpiled code will work in ANY JavaScript environment, be it Node.js or the browser. Further, this application was published to the *NPM(Node Package Manager)* registry so that anybody can use it as an NPM package in their Node.js applications. Also, a *CDN (Content Delivery Network)* copy was hosted on *jsDelivr*. This allows anyone to use our application in their websites.

Now one might ask what was the need for transpiling ES6 code to CJS, when we could have written the code in the CJS syntax itself. The point is to make our code *future-proof*.

## 1.1 Demonstration in Browser Environment

### Demo

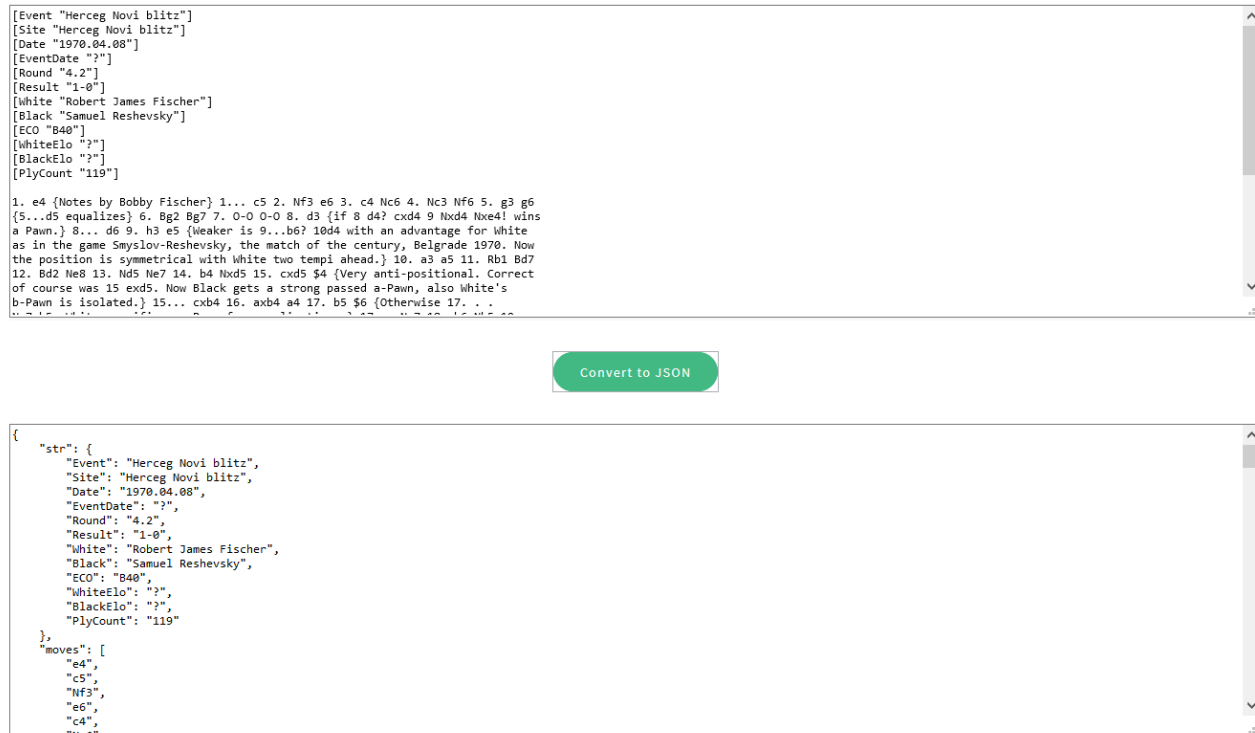


Figure 3: Demo at: <https://aditya-ds-1806.github.io/Chess-PGN-Parser/>

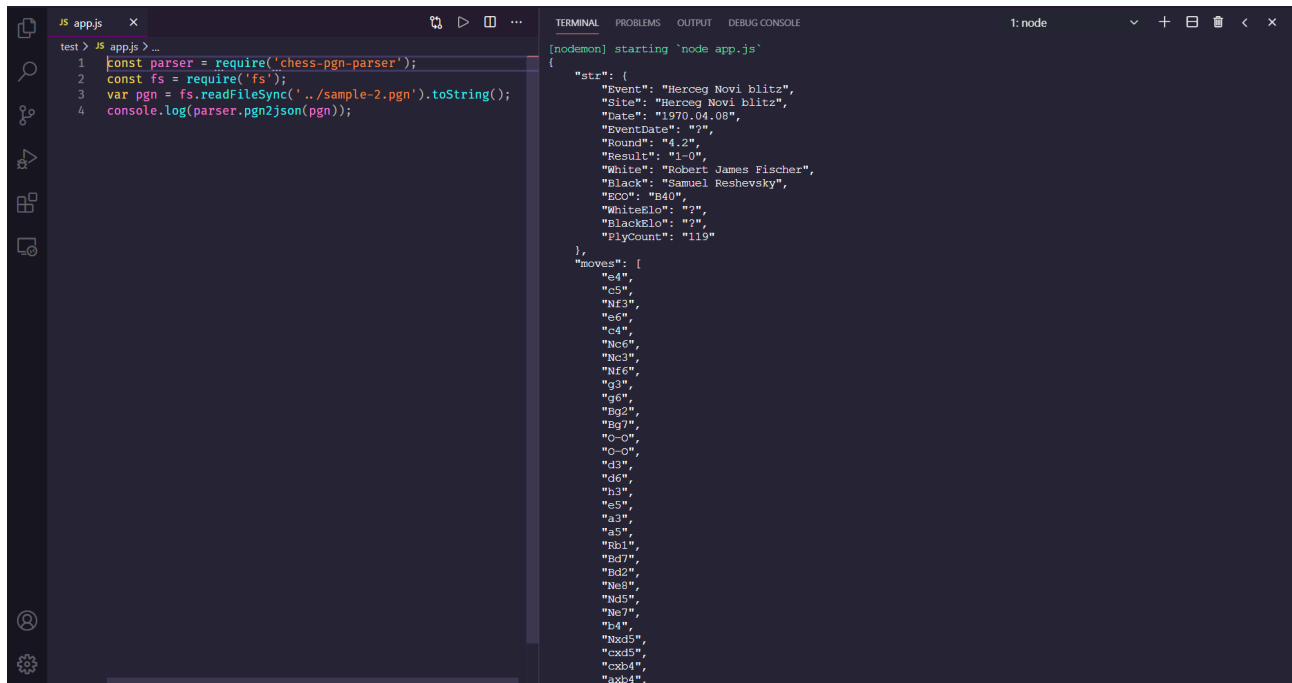
A documentation website was also built for the benefit of developers. The documentation website also has a live demo wherein a user can paste a PGN file's contents and get the JSON output or convert the default PGN text. As was mentioned previously, a CDN was used to include the script file in the website. Figure 3 shows the code in action on the website.

## 1.2 Demonstration in Node.js Environment

### 1.2.1 Driver Code

```
const parser = require('chess-pgn-parser');
const fs = require('fs');
var pgn = fs.readFileSync(' ../sample-2.pgn').toString();
console.log(parser.pgn2json(pgn));
```

### 1.2.2 Output

The image shows a screenshot of a VS Code editor. On the left, a file named 'app.js' is open, containing the following code:

```
1 const parser = require('chess-pgn-parser');
2 const fs = require('fs');
3 var pgn = fs.readFileSync(' ../sample-2.pgn').toString();
4 console.log(parser.pgn2json(pgn));
```

On the right, the 'TERMINAL' pane is active, showing the output of the command 'node app.js'. The output is a JSON object representing a chess game. The first part of the output is the game header, and the second part is an array of moves. The moves shown are: e4, e5, Nf3, e6, c4, Nc6, Nc3, Nf6, g3, g6, Bg2, Bg7, O-O, O-O, d3, d6, h3, e5, a3, a5, Bb1, Bd7, Bd2, Ne8, Nd5, Ne7, Bb4, Nxd5, cxd5, cxb4, axb4.

Figure 4: A portion of the terminal output that shows the moves

Figures 4 and 5 show the JSON output in the console on the right and the driver code on the left. All of the output could not be captured in a single image. Figures 4 and 5 show different parts of the output.

```

1 const parser = require('chess-pgn-parser');
2 const fs = require('fs');
3 var pgn = fs.readFileSync('.../sample-2.pgn').toString();
4 console.log(parser.pgn2json(pgn));

```

```

{
  "moveCount": 8,
  "comment": "if 8 d4? cxd4 9 Nxd4 Nxe4! wins a Pawn."
},
{
  "moveCount": 9,
  "comment": "Weaker is 9...b6? 10d4 with an advantage for White as in the game Smyslov-Rashevsky, t
he match of the century, Belgrade 1970. Now the position is symmetrical with White two tempi ahead."
},
{
  "moveCount": 15,
  "comment": "Very anti-positional. Correct of course was 15 exd5. Now Black gets a strong passed a-
Pawn, also White's b-Pawn is isolated."
},
{
  "moveCount": 17,
  "comment": "Otherwise 17. . . Nc7-b5. White sacrifices a Pawn for complications."
},
{
  "moveCount": 21,
  "comment": "If 20 Rxa4?,... Nc3 wins the exchange."
},
{
  "moveCount": 25,
  "comment": "25. . . d3! won by force. For example a) 26 Bd7 Bal 27 Rxb6 Bb2 or . . . Bd4 etc. b) 2
6 e5 Rd2! 27 Nd2 Be5 28 Bd7 Bal 29 Rb6 Bd4 etc."
},
{
  "moveCount": 31,
  "comment": "A last swindle try."
},
{
  "moveCount": 31,
  "comment": "31. . . Ra2! won easily 32.Qxa2 Qel+ 33.Kg2 d2, But as Dr. Tarrasch said \"You must
see it\""}
},
{
  "nag": [
    {
      "moveCount": 15,
      "value": "$4"
    },
    {
      "moveCount": 17,
      "value": "$6"
    },
    {
      "moveCount": 21,

```

Figure 5: A portion of the terminal output that shows the annotations and NAGs

## 2 Parsing large PGN files that contain multiple games

Now that the PGN Parser is ready, we can now proceed towards benchmarking the multi-threaded and single-threaded approaches.

### 2.1 Single-threaded Approach

The naive approach would be to simply parse the whole PGN file and extract individual games and use the `pgn2json` method on each individual game using a loop as shown below in the `getJSONWithoutWorker` method. The results are then appended to a JSON file.

```

exports.getJSONWithoutWorker = function (pgns) {
  var obj = pgns.map(pgn => JSON.parse(pgn2json(pgn)));
  fs.writeFileSync('games.json', JSON.stringify(obj, null, 4));
}

```

### 2.2 Multi-threaded Approach

In the multi-threaded approach, we use some of Node.js' in-built functionalities to determine the number of CPU cores available on the system in use. Based upon this, we distribute the task of converting each individual game to JSON among these cores (which consist of threads) using the built-in `worker_threads` module. The `getJSONWithWorker` method shown below does exactly this. Each worker thread is given the code to execute and the data to be processed. Each worker then runs the `pgn2json` method on the games it has received and the results are appended to a JSON file.

```

exports.getJSONWithWorker = function (pgns) {
  const segmentSize = Math.ceil(pgns.length / userCPUCount);
  const segments = [];

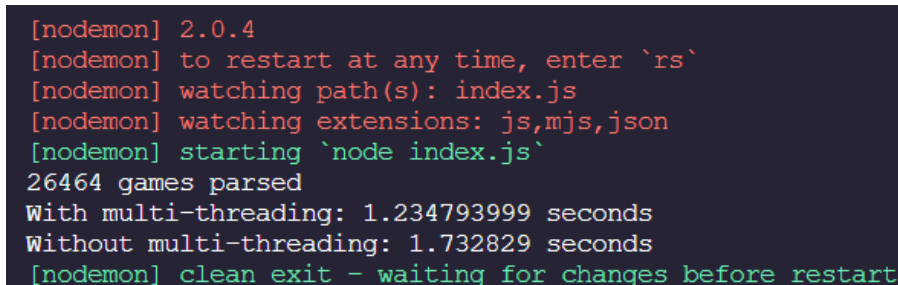
  for (let segmentIndex = 0; segmentIndex < userCPUCount; segmentIndex++) {
    const start = segmentIndex * segmentSize;
    const end = start + segmentSize;
    const segment = pgns.slice(start, end);
    segments.push(segment);
  }

  var promises = segments.map(
    segment =>
      new Promise((resolve, reject) => {
        const worker = new Worker(workerPath, {
          workerData: segment,
        });
        worker.on('message', resolve);
        worker.on('error', reject);
        worker.on('exit', code => {
          if (code !== 0) reject(new Error('Worker stopped with
            exit code ${code}'));
        });
      })
  );

  return Promise.all(promises).then(objects => {
    var jsonGames = [];
    objects.forEach(object => jsonGames.push(object));
    fs.writeFileSync('games-threaded.json', JSON.stringify(jsonGames,
      null, 4));
  });
}

```

## 2.3 Demonstration



```

[nodemon] 2.0.4
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): index.js
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
26464 games parsed
With multi-threading: 1.234793999 seconds
Without multi-threading: 1.732829 seconds
[nodemon] clean exit - waiting for changes before restart

```

Figure 6: Output for roughly 13000 games

For the purpose of demonstration, a PGN file containing 827 games was downloaded from the internet and it's contents were duplicated multiple times to test the performance of each approach for different PGN file sizes. The above image shows the results for a PGN file containing roughly 13000 games. The JSON output is shown below:

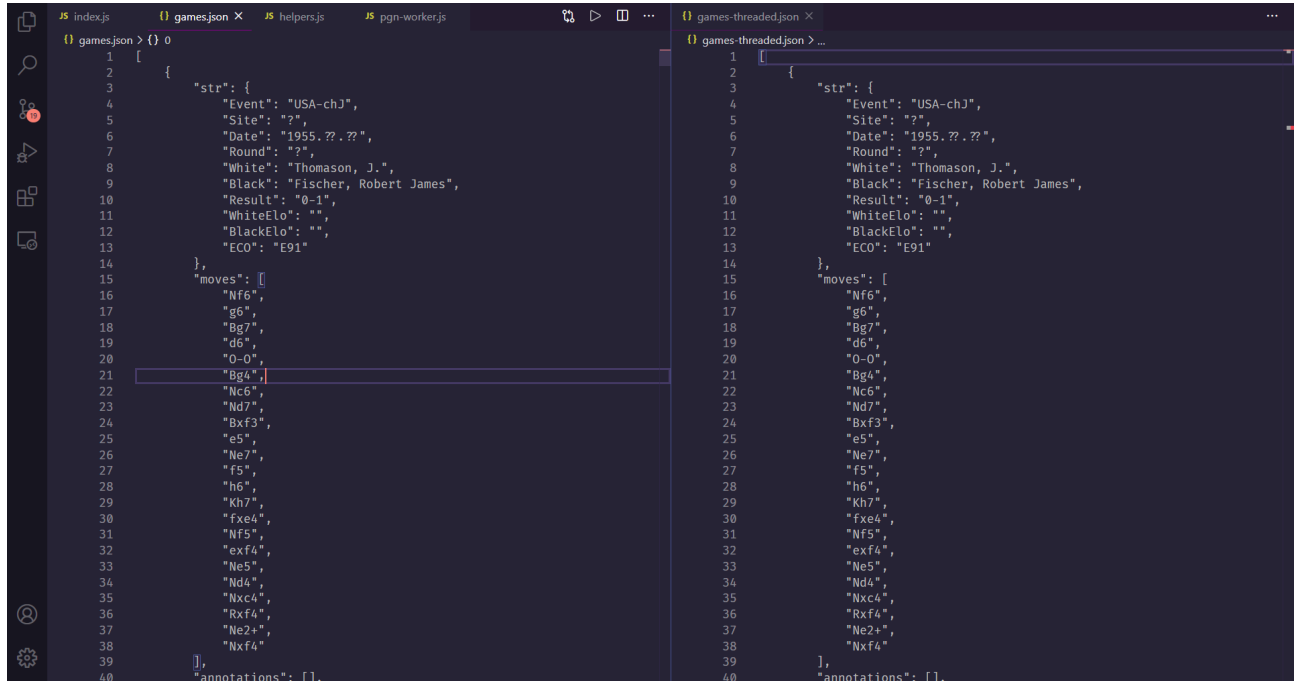


Figure 7: A glimpse of the output after parsing 13000 games. On the left is the output for single-threaded version and on the right is the output for the multi-threaded version.

### 3 Benchmarking

The `benchmark` function is used to compare the two methods: `getJSONWithWorker` and `getJSONWithoutWorker`. It uses a the `hrtime` method which returns a very precise *high-resolution real time in a [seconds, nanoseconds] tuple Array*, where *nanoseconds* is the remaining part of the real time that can't be represented in second precision. The time taken to run each method is then printed in the console as shown previously in Figure 6.

```
exports.benchmark = async function (func, arg, label) {
  var start, diff;
  start = process.hrtime();
  await func(arg);
  diff = process.hrtime(start);
  console.log(`${label}: ${diff[0] + diff[1] / NS_PER_SEC} seconds`);
}
```

## 4 Results

The results are summarised below. The benchmarking was done on an Intel i7 8th generation processor with 8 cores:

Trial	Multi-threaded(s)	Single-threaded(s)
0.	0.1875034	0.0635169
1.	0.192108101	0.145554001
2.	0.150625801	0.066774601
3.	0.181777899	0.0724916

Table 1: Time taken to fully parse 827 games and output JSON

Trial	Multi-threaded(s)	Single-threaded(s)
0.	0.214608	0.1311382
1.	0.2279158	0.165930699
2.	0.176748599	0.116851299
3.	0.2525838	0.228234901

Table 2: Time taken to fully parse 1654 games and output JSON

Trial	Multi-threaded(s)	Single-threaded(s)
0.	0.2023168	0.1913543
1.	0.2652761	0.2286863
2.	0.274466101	0.2360691
3.	0.250089801	0.235808

Table 3: Time taken to fully parse 3308 games and output JSON

Trial	Multi-threaded(s)	Single-threaded(s)
0.	0.330833499	0.361550699
1.	0.3150702	0.360780701
2.	0.3071193	0.3648517
3.	0.309957799	0.369372501

Table 4: Time taken to fully parse 6616 games and output JSON

Trial	Multi-threaded(s)	Single-threaded(s)
0.	0.5780579	0.712450199
1.	0.622088299	0.7036642
2.	0.5669147	0.6878104
3.	0.6043357	0.698677301

Table 5: Time taken to fully parse 13232 games and output JSON

Trial	Multi-threaded(s)	Single-threaded(s)
0.	1.1900514	2.117208501
1.	1.2130323	1.7196827
2.	1.2297011	1.741066099
3.	1.2856137	1.6399716

Table 6: Time taken to fully parse 26464 games and output JSON

Trial	Multi-threaded(s)	Single-threaded(s)
0.	2.996129099	3.862680001
1.	2.594330399	3.5656342
2.	3.1488285	3.507052801
3.	3.0087395	3.8729383

Table 7: Time taken to fully parse 52928 games and output JSON

Trial	Multi-threaded(s)	Single-threaded(s)
0.	5.916030901	8.091521899
1.	5.7170082	8.637040599
2.	5.8032977	7.7143801
3.	5.7086765	7.815089801

Table 8: Time taken to fully parse 105856 games and output JSON

Table 9 summarises the results from all previous tables. It tabulates the average time taken to parse PGN files of varying sizes for both the approaches discussed so far.

Games	Multi-threaded(s)	Single-threaded(s)
827	0.17800380025	0.0870842755
1654	0.21796404975	0.16053877475
3308	0.2480372005	0.222979425
6616	0.3157451995	0.36413890025
13232	0.59284914975	0.700650525
26464	1.229599625	1.804482225
52928	2.9370068745	3.7020763255
105856	5.78625332525	8.06450809975

Table 9: Table summarising the average time taken to parse PGN files of varying sizes for single-threaded and multi-threaded approaches.

The data from Table 9 was used to generate the following plot in Excel.

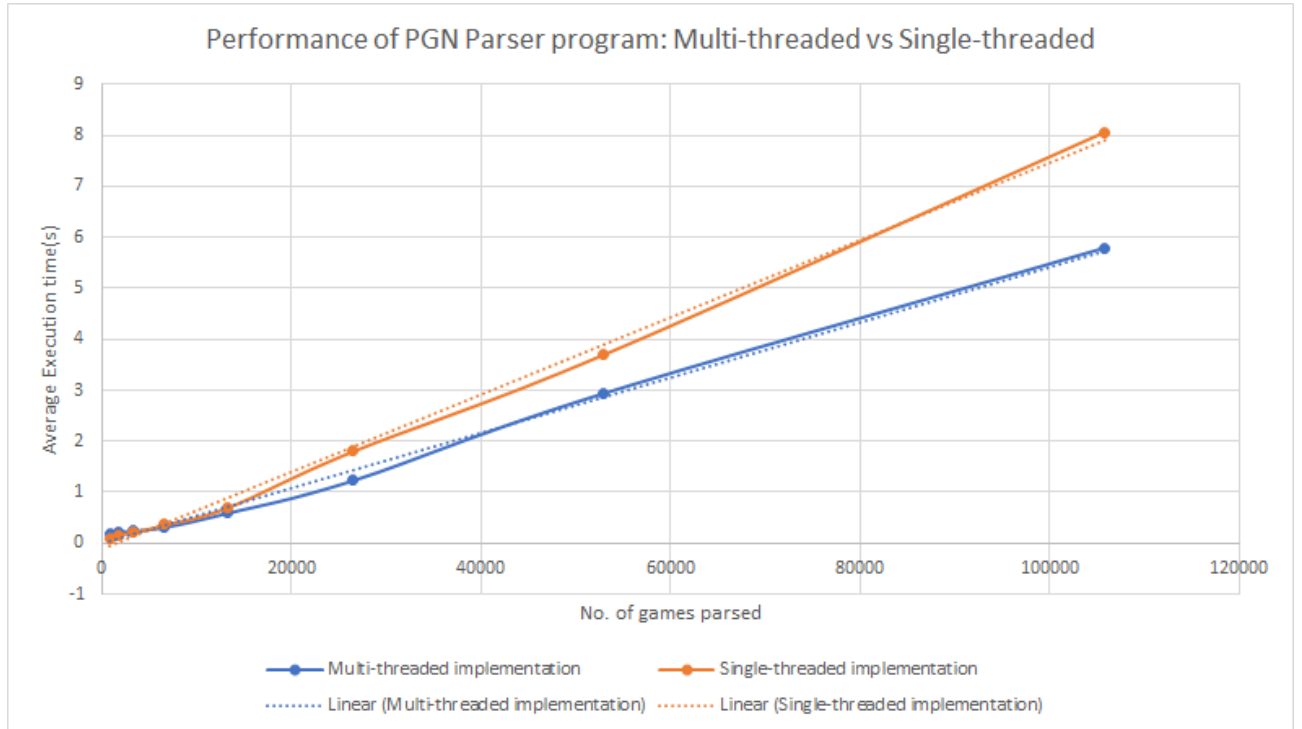


Figure 8: Plot of games parsed vs the average time taken

## 5 Conclusion

From Table 9, we observe that multithreading is an overkill for PGN files that contain anything less than around 5,000 games which is quite expected. In fact, the multi-threaded version performs slightly worse than the single-threaded version.

However, for files that contain more than 5,000 games, the multithreaded version really starts to take over. For 100,000 games, the execution time for both the approaches differ by close to



3 seconds! Another interesting thing that can be seen from Figure 8 is that the variations are almost linear which wasn't something we expected.

Thus, we conclude that, for smaller PGN files the multi-threaded approach doesn't bring any major performance benefits. But as the file size grows, *the multi-threaded approach outperforms the single-threaded approach*.

## 6 Important Links

### 6.1 Parser

0. NPM registry: <https://www.npmjs.com/package/chess-pgn-parser>
1. GitHub: <https://github.com/Aditya-ds-1806/Chess-PGN-Parser>
2. Documentation Website: <https://aditya-ds-1806.github.io/Chess-PGN-Parser>
3. jsDelivr CDN: <https://www.jsdelivr.com/package/npm/chess-pgn-parser>

### 6.2 Project

0. GitHub: <https://github.com/Aditya-ds-1806/OS-Project>