Intro to Parallel Programing

Term Project Report

By Tim Inzitari

**Introduction**

For my term project I was tasked with designing a parallel solution for a dataset of my interest. The dataset that was chosen was titled “Chess Game Dataset (Lichess)” on Kaggle.com [1] by Mitchell J. This dataset contains just over 20,000 chess games from users on the website Lichess.org. The problem we chose to design an answer to and solve was finding the average rating of the players and count the number of games that were able to be recorded with a non-zero elapsed time. After designing these solutions in parallel we evaluated the speedup and efficiency of our solution over various subsets of the data. All iterations of the program were run on an AMD Ryzen 9 3900X 12-Core Processor with standard clock rate of 3.80GHz.

**Dataset and Algorithm**

The “Chess Game Dataset” contained 16 points of data for every entry in the set. These entries were: Game ID, whether the game was rated or unrated, start time, end time, turn count, description of how the game ended, who won the match, the time control used, the id of the white player, the rating of the white player, the id of the black player, the rating of the black player, the moves of the game in standard chess notation, the opening eco, the opening name, and the number of moves in the opening phase. For this project we specifically used the start time, end time, number of turns, and white player rating in our algorithms.

A significant problem with the dataset that was chosen fell in the categories of start time and end time. These values were stored as large integers of milliseconds that surpassed 1012 in scientific notation, however the set only stored 6 significant digits, so any games that took less than 16 minutes and 40 seconds were registered in the dataset as having taken no time at all. This caused us to alter our original plan of finding average game length at certain turn counts to finding the amount of games that simply took any time at all.

The algorithms we designed handled two separate tasks involving the data, the first was finding how many games registered a non-zero time for games of at least a certain turn count. The second algorithm we implemented was finding the average rating of the white player for games of at least a certain turn count. For both of these algorithms we first started by created the subset of only entries that had at least the number of turns the user requested.

After creating the subset, the first algorithm then began to search them for games that had an elapsed time that registered as non-zero. This was done by splitting up our newly created subset among the processors and finding local counts and what the local max length was. After our local values were found we entered a critical section where we added the global count and found a global maximum.

After creating the subset, the second algorithm began to find the sum of all white player ratings for the entries in the subset. This was done by splitting up the newly created subset among the processers and finding local sums for the regions allocated to them. These local sums then entered a critical section to be added to a global sum. The resulting global sum was then divided by the number of entries in the subset to find the average white player rating for that set.

**Implementation**

When implementing this project, we had several options that we could use for programming language and parallel interface that we could use. We decided to use C++ as the programming language of choice and utilized the POSIX-thread (pthreads) parallel interface. We also utilized the mutex structure when handling critical sections of the algorithms.   
 We read the data from the csv file given to us off Kaggle and stored each entry as a Class we entitled “ChessGame”. We stored the entries in a vector of chess games. This was done so we could easily access, add, and edit parts of the dataset easily.

To aid the evaluation of speedup and efficiency for our solution we also recorded the time it took to do each algorithm for the various dataset sizes.

**Results and Analysis**

When determining the datasets sizes, we tested over, it was decided to test various turn count lower limits. The limits chosen were: 5, 10, 20, 40, and 80. This resulted in the following dataset sizes:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Lower Limit** | 5 | 10 | 20 | 40 | 80 |
| **Dataset Size** | 19716 | 19441 | 18397 | 14383 | 4900 |

We also tested using various thread-counts, the thread counts we chose were 1, 2, 4 and 8.

*Table 1* were the time results for the first algorithm that found the maximum length and number of non-zero timed games in seconds:

*Table 1*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Times | **19716** | **19441** | **18397** | **14383** | **4900** |
| **1** | 2.09E-04 | 1.20E-04 | 1.30E-04 | 1.11E-04 | 7.99E-05 |
| **2** | 1.53E-04 | 1.39E-04 | 1.24E-04 | 8.39E-05 | 7.10E-05 |
| **4** | 1.34E-04 | 1.35E-04 | 1.67E-04 | 1.16E-04 | 1.13E-04 |
| **8** | 2.37E-04 | 3.09E-04 | 2.42E-04 | 2.28E-04 | 2.40E-04 |

*Formula 1*

*Formula 2*

We apply *Formula 1* on *Table 1* we find the speed up values for our first algorithm’s parallel solution. They are shown in *Table 3*:

*Table 2*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **SPEEDUP** | **19716** | **19441** | **18397** | **14383** | **4900** |
| **1** | 1 | 1 | 1 | 1 | 1 |
| **2** | 1.364486 | 0.862779 | 1.048076 | 1.323864 | 1.124161 |
| **4** | 1.55872 | 0.888692 | 0.778571 | 0.958848 | 0.706751 |
| **8** | 0.881288 | 0.388117 | 0.536946 | 0.487448 | 0.332671 |

When we apply *Formula 2* on *Table 1* we find the following efficiency values for our first algorithm’s parallel solution. They are shown in *Table 3*:

*Table 3*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **EFFICIENCY** | **19716** | **19441** | **18397** | **14383** | **4900** |
| **1** | 1 | 1 | 1 | 1 | 1 |
| **2** | 0.682243 | 0.431389 | 0.524038 | 0.661932 | 0.56208 |
| **4** | 0.38968 | 0.222173 | 0.194643 | 0.239712 | 0.176688 |
| **8** | 0.110161 | 0.048515 | 0.067118 | 0.060931 | 0.041584 |

*Table 4* is the time results for the second algorithm that found the average rating of the white player in the data subset in seconds:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Times | **19716** | **19441** | **18397** | **14383** | **4900** |
| **1** | 2.23E-04 | 1.73E-04 | 1.71E-04 | 1.58E-04 | 9.89E-05 |
| **2** | 1.29E-04 | 1.11E-04 | 1.13E-04 | 1.03E-04 | 1.06E-04 |
| **4** | 1.89E-04 | 1.32E-04 | 1.29E-04 | 1.16E-04 | 1.13E-04 |
| **8** | 5.51E-04 | 2.32E-04 | 2.57E-04 | 2.72E-04 | 2.98E-04 |

*Table 4*

We apply *Formula 1* on *Table 4* we find the speed up values for our second algorithm’s parallel solution. They are shown in *Table 5*:

*Table 5*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **SPEEDUP** | **19716** | **19441** | **18397** | **14383** | **4900** |
| **1** | 1 | 1 | 1 | 1 | 1 |
| **2** | 1.730129 | 1.561291 | 1.512658 | 1.534722 | 0.9346851 |
| **4** | 1.180328 | 1.310469 | 1.325323 | 1.361397 | 0.8755275 |
| **8** | 0.40502 | 0.74538 | 0.665121 | 0.581069 | 0.332 |

When we apply *Formula 2* on *Table 4* we find the following efficiency values for our first algorithm’s parallel solution. They are shown in *Table 6*:

*Table 6*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **EFFICIENCY** | **19716** | **19441** | **18397** | **14383** | **4900** |
| **1** | 1 | 1 | 1 | 1 | 1 |
| **2** | 0.865064 | 0.780645 | 0.756329 | 0.767361 | 0.4673426 |
| **4** | 0.295082 | 0.327617 | 0.331331 | 0.340349 | 0.2188819 |
| **8** | 0.050627 | 0.093172 | 0.08314 | 0.072634 | 0.0415 |

As for the results of our algorithms, the game counts found in the first algorithm can be found in *Table 7*, the maximum times (in milliseconds) found from the first algorithm can be found in *Table 8*, and the average ratings found from the second algorithm can be found in *Table 9*.

*Table 7*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Lower Turn Limit** | **5** | **10** | **20** | **40** | **80** |
| **Non-Zero Timed Games** | 11271 | 11141 | 10579 | 8442 | 2998 |

*Table 8*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Lower Turn Limit** | **5** | **10** | **20** | **40** | **80** |
| **Max Game Time** | 6.058447e+08 | 6.058447e+08 | 6.058447e+08 | 9.0e+07 | 1.035103e+07 |

*Table 9*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Lower Turn Limit** | **5** | **10** | **20** | **40** | **80** |
| **Average Rating** | 1599.03 | 1600.82 | 1606.65 | 1616.19 | 1635.87 |

Using Tables 1-6 we can analyze the scalability of our problem and the solution we implemented. Based on the results for both algorithms we can find that neither algorithm was scalable, as time increased as we varied thread sizes in most cases. This mainly may be due to the critical section requirements in both algorithms that will slow down the solution. The larger data subsets are much closer to being scalable but still not at the point that it can be considered scalable. Overall, the best solution for these problems would be a serial one for datasets of this size.  
 We also noticed that a potential relationship between turn count and rating could exist. When ever we increased our lower limit on turn count, the average rating increased.

**Conclusion**

In this project we analyzed the scalability of two designed solutions for the “Chess Game Dataset (Lichess)” dataset. Neither of our solutions were scalable regarding this dataset, however larger subsets did provide better results. This could lead to further exploration of a larger dataset on the subject. This also could be due to limitations in the Pthread architecture, and the requirements set by a mutex critical section. Further points of research could be using different critical section architectures and different parallel APIs.

Another result that we found in this project that could lead to further research is the potential link between turn count and player rating that we found. Based on our results a loose relationship could exist. Further research could involve developing a regression equation for the dataset using machine-learning algorithms.

**References**

**[1]**

J, M. (2017). *Chess Game Dataset (LiChess)*. Retrieved from Kaggle: https://www.kaggle.com/datasnaek/chess