

Lohith Maralla
12/08/22

Abstract

This project attempted to solve the problem of calculators' slow computation time with complex expressions. This was an important problem to address because a faster and more efficient calculator can save time for the user. The problem was addressed by creating a calculator that utilized threads to compute expressions faster. Utilizing threads to create a calculator allows certain operations to be computed simultaneously instead of being computed individually, and as a result, decreasing waiting and computational time.

Introduction and Background

There are many existing calculators on the web but certain calculators differentiate themselves from others based on their faster computational time and wider array of operations they can execute. This problem was important to address since certain calculators cannot process a very complex expression with very large numbers such as a simple four-function calculator. This problem, however, has been addressed by calculators such as Wolfram Alpha. Similar problems in this area that have been addressed are computing word counts of documents. Our approach to utilizing threads in our algorithm makes our calculator creative. Specifically, compared to other multithreaded calculators that exist, our calculator creates a new process for every operation rather than grouping numbers in an arithmetic expression and creating a new process. In order to solve the problem, we analyzed previous multithreaded calculators to get inspiration and to see if a more unique calculator could be created. In addition, we had to understand how to utilize threads and other programming constructs. For example, we had to understand how to utilize locks and mutexes to prevent race conditions from occurring in our algorithm.

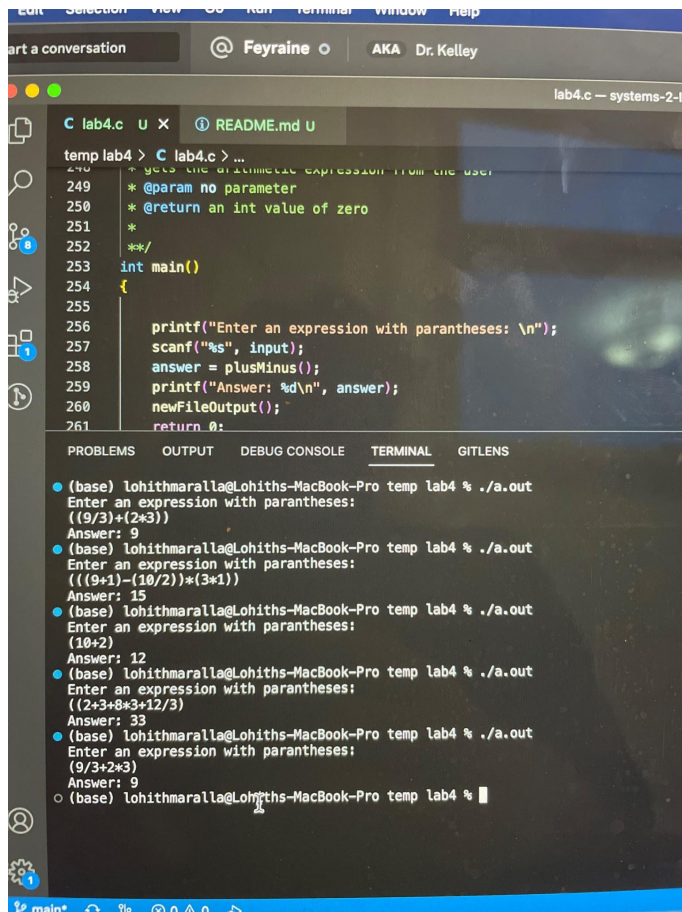
Methodology and Implementation

We took inspiration from Nizam Alešević's (2018) multithreaded calculator to see how a multithreaded calculator could be created. I, Lohith Maralla, wrote lines 1-25, 51-75 (functions that utilize threads to multiply/divide numbers), 113-147 (functions to create multiplication and division threads), 149-192 (functions to parse and evaluate expressions), and 225-245 (function to write output to a file). Matthew Kuriakose wrote lines 25-49 (functions that utilize threads to add/subtract numbers), lines 75-111 (functions to create addition and subtraction threads), lines 194-223 (function that parsed through the expression), and 247-262 (main method). The code addresses the problem by creating threads for every operation in the algorithm. The algorithm takes user input for an arithmetic expression and parses through the expression starting in the plusMinus function. The plusMinus calls the timesDiv function which calls the para function to follow PEMDAS and to parse through the expression properly. The digits contained in the expression are returned by para() to the timesDiv function. If the arithmetic expression contains multiplication or division, timesDiv() calls multiCompute/diviCompute to create a thread process for the multiplication/division operation. Within these functions, mutex locks are set and a thread is created that computes the operation. The mutex locks protect the critical section within the

function which is where the thread is created to compute the operation. This prevents a race condition from occurring and other threads accessing the same operation. The mutexes help prevent the need of the process control switch which saves some time. Similarly, this is also implemented in the addition and subtraction functions. We had a difficult time implementing the mutexes and resolved this issue by going to office hours. Normally an expression would compute operations in respective order. However, this algorithm utilizes data parallelism in certain situations, such as $((6+4+(4*5)))$, enabling for the multiplication/division threads to compute synchronously with the addition/subtraction threads since these threads are computing simultaneously. When data parallelism is being utilized this program is able to decrease computational time and handle larger inputs.

Results

There were many issues faced while coding. For instance, our code does not work without the use of parentheses, and we addressed this issue by leaving a note and an example in the README to input parentheses when giving input to the console. In addition, our code was expected to successfully run on all platforms. Our code failed to run some complex expressions on Replit and Stdlinux but was able to successfully run all computations on Vscode. While coding our algorithm, we ran into a few issues. One major issue that kept breaking our code was failing to join the threads that were created. This was fixed by using `pthread_join` in the appropriate places. Another issue was utilizing struct pointers incorrectly. This created issues and was solved by correctly using a pointer to the struct when necessary such as in the functions that computed an operation for the threads. Here is an illustration of the code running and its output.



The screenshot shows a Visual Studio Code editor with a C file named `lab4.c`. The code is a simple calculator program that prompts the user for an arithmetic expression, evaluates it, and prints the result. The code includes comments and a `main` function. Below the code, the terminal window shows the program's execution. The user enters several expressions, and the program outputs the corresponding results.

```
temp lab4 > C lab4.c > ...
248 // gets the arithmetic expression from the user.
249 * @param no parameter
250 * @return an int value of zero
251 *
252 */
253 int main()
254 {
255
256     printf("Enter an expression with parantheses: \n");
257     scanf("%s", input);
258     answer = plusMinus();
259     printf("Answer: %d\n", answer);
260     newFileOutput();
261     return 0;
}

(base) lohithmaralla@Lohiths-MacBook-Pro temp lab4 % ./a.out
Enter an expression with parantheses:
((9/3)+(2*3))
Answer: 9
(base) lohithmaralla@Lohiths-MacBook-Pro temp lab4 % ./a.out
Enter an expression with parantheses:
(((9+1)-(10/2))*(3+1))
Answer: 15
(base) lohithmaralla@Lohiths-MacBook-Pro temp lab4 % ./a.out
Enter an expression with parantheses:
(10+2)
Answer: 12
(base) lohithmaralla@Lohiths-MacBook-Pro temp lab4 % ./a.out
Enter an expression with parantheses:
((2+3+8*3+12/3))
Answer: 33
(base) lohithmaralla@Lohiths-MacBook-Pro temp lab4 % ./a.out
Enter an expression with parantheses:
(9/3+2*3)
Answer: 9
(base) lohithmaralla@Lohiths-MacBook-Pro temp lab4 %
```

Conclusion

If given the chance to repeat this project, we would make the program more consistent. The program should be able to run on all platforms successfully and should utilize data parallelism in all situations. In addition, the algorithm should be able to deal with any type of user input such as incorrect user input. The project went alright considering the time frame, but if given more time, we would hope to make the program more reliable and consistent. The algorithm to parse through the arithmetic expression without the threads was nice so that code could be used for a future arithmetic solver. My partner and I handled the workload between us well; we were able to split up the work evenly and communicated well.

References

Alešević, Nizam. "Multithreaded-Calculator." 2018.
<https://github.com/nalesevic/Multithreaded-Calculator>.