Exercise Set 3

Darina Öö

05.02.2024

1 Problem 1

1.1 Ackermann Function Implementation

The Ackermann function is defined recursively as follows:

```
int Ackermann(int m, int n) {
    if (m == 0) return n + 1;
    if (n == 0) return Ackermann(m - 1, 1);
    return Ackermann(m - 1, Ackermann(m, n - 1));
}
```

The main function calls the Ackermann function with predefined values for m and n:

```
int main() {
    int result = Ackermann(4, 1);
    std::cout << "Ackermann(4, 1) = " << result << std::endl;
    return 0;
}
```

1.2 Compilation

The program was compiled on an Ubuntu system with debugging symbols enabled:

```
g++ -g -o ackermann ackermann.cpp
```

1.3 Debugging

The program was run under the GNU Debugger (gdb) and was interrupted to inspect the call stack:

```
(gdb) run
(gdb) where
```

1.4 Analysis

During the debugging session, the program encountered a segmentation fault, indicative of a stack overflow. This error occurs because the Ackermann function's deep recursion exceeds the system's allocated stack size for the process. The output from the 'where' command in gdb confirmed the recursive nature of the function calls, as shown by the following terminal output:

```
Program received signal SIGINT, Interrupt.
0x0000aaaaaaaa0a7c in Ackermann (m=1, n=29878) at ackermann.cpp:10
10     return Ackermann(m - 1, Ackermann(m, n - 1));

(gdb) where
#0  0x0000aaaaaaaa0a7c in Ackermann (m=1, n=29878) at ackermann.cpp:10
#1  0x0000aaaaaaaa0a7c in Ackermann (m=1, n=29879) at ackermann.cpp:10
...
```

This output indicates that at the point of interruption, the program was executing a deeply nested recursive call with m reduced to 1 and n increased to nearly 30000. Such deep recursion is typical for the Ackermann function, which is known for its extreme growth rate.

The stack trace shows that the Ackermann function is called with $m = 1$, and n is a large value, which suggests that the function was in the process of recursively reducing m to 0, at which point it would begin to return through the stack frames. However, due to the recursion depth, the stack frames required exceed the maximum stack size, leading to a segmentation fault.

This exercise demonstrates the practical limitations of recursion in software, particularly for functions like the Ackermann function, which can quickly grow beyond system resource limits.

2 Problem 2

2.1 Makefile Modifications

To enable profiling with 'gprof', the Makefile was modified to include the '-pg' flag in both the compilation and linking processes. The original compilation flags included '-O2' for optimization. The modifications were made as follows:

```
# Original CFLAGS
CC=gcc -O2

# Modified CFLAGS with profiling enabled
CC=gcc
CFLAGS=-O2 -pg
LDFLAGS=-lm -pg

# Original linking command
```

```
$(CC) -o $(TARGET) $(OBJECTS) -lm

# Modified linking command with profiling enabled
$(CC) $(CFLAGS) -o $(TARGET) $(OBJECTS) $(LDFLAGS)
```

These changes instruct the GCC compiler to include additional debugging information necessary for ‘gprof‘ and link the program with the profiling library. The ‘-pg‘ option generates extra code to write profile information suitable for the analysis by ‘gprof‘.

2.2 Makefile Compilation Commands

Each object file compilation command in the Makefile was also updated to use the new ‘CFLAGS‘.

```
# Original compilation command for forces.c
$(CC) -c forces.c

# Modified compilation command with profiling enabled
$(CC) $(CFLAGS) -c forces.c
```

The ‘clean‘ target in the Makefile was left unchanged, as it is used to remove the object files and the executable, which is not affected by the addition of profiling flags.

2.3 Profiling Compilation

After modifying the Makefile, the code was recompiled to produce an executable capable of generating profiling data. The following commands were executed in the terminal:

```
make clean
make
```

This process ensures that all object files and the executable are built with profiling information embedded, which ‘gprof‘ can later use to analyze the program’s performance.

The Makefile was modified to include profiling flags. The code was compiled with the following commands:

```
ln -s ../input/atoms.in
ln -s ../input/mdmorse.in
```

After successful compilation, the executable ‘mdmorse‘ was run, and profiling data was collected:

```
./mdmorse
```

```
gprof mdmorse gmon.out > profiling_analysis.txt
```

The profiling output indicated that the most CPU time-intensive operations occurred in the ‘GetForces‘ and ‘UpdateNeighbourlist‘ functions.

2.4 Function with Most CPU Time

The ‘GetForces‘ function was identified as the most time-consuming part of the code, consuming 63.12% of the CPU time. This function is called 30,000 times during the execution, indicating its critical role in the simulation's performance.

2.5 Second Most CPU-Intensive Function

The second most CPU time was spent in the ‘UpdateNeighbourlist‘ function, accounting for 33.95% of the total CPU time with 6,000 calls.. This function was called 6,000 times, suggesting its significance in the overall performance of the simulation.

2.6 Conclusions

The profiling analysis revealed that the ‘GetForces‘ function is the primary consumer of CPU time in the molecular dynamics simulation code. Optimization efforts should focus on this function to improve the overall efficiency of the simulation. The detailed profiling data underscores the need for computational efficiency in simulations that perform large numbers of force calculations.

3 Problem 3

The program ‘ex3p3.c‘ was compiled with GCC using four different optimization flags. The execution time for each compiled version was measured using the Unix ‘time‘ command, focusing specifically on the loop's performance.

3.1 Compilation

The program was compiled with the following optimization flags:

- No optimization: `gcc -O0 -o ex3p3_O0 ex3p3.c`
- Basic optimization: `gcc -O1 -o ex3p3_O1 ex3p3.c`
- Further optimization: `gcc -O2 -o ex3p3_O2 ex3p3.c`
- Full optimization: `gcc -O3 -o ex3p3_O3 ex3p3.c`

Each compiled version was executed with these commands:

```
time ./ex3p3_O0
time ./ex3p3_O1
time ./ex3p3_O2
time ./ex3p3_O3
```

3.2 Execution

Consequently, the following execution times were recorded:

Optimization Level	Real Time (s)	User Time (s)	System Time (s)
-O0	10.425	1.117	9.301
-O1	2.291	0.316	1.972
-O2	1.306	0.210	1.093
-O3	1.260	0.041	1.217

3.3 Analysis and Discussion

The results indicate a clear trend: as the level of optimization increases, both the real and user time decrease significantly. The most substantial gain is observed when moving from -O0, which has no optimization, to -O1. This improvement continues, albeit at a decreasing rate, as the optimization level increases to -O2 and -O3.

Interestingly, while the user time decreases consistently with higher optimization levels, the system time fluctuates. This fluctuation may be due to various factors not directly related to the computation, such as operating system scheduling, I/O operations, and system calls.

The optimization level -O3 offers the best performance in terms of user time, suggesting that the compiler's aggressive optimizations lead to much faster code execution. However, the real-time improvement from -O2 to -O3 is marginal, indicating diminishing returns for the most aggressive optimization level.

These findings underscore the importance of compiler optimizations in high-performance computing and the need to balance the benefits against the potential risks of aggressive optimization, such as increased compilation time or possible changes in program behavior.

4 Problem 4

4.1 Program Description and execution

The program performs a large number of mathematical calculations within a loop and writes the results to a file. It measures both the CPU time using the 'clock()' function and the elapsed time using the 'gettimeofday()' function.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <sys/time.h>
#include <math.h>

#define N 15000
int a[N][N];
```

```
#define NPRINT 500

int main() {
  int i, j;
  clock_t t1, t2;
  struct timeval start, end;
  double cpu_time_used, elapsed_time;

  /* Begin measurement */
  t1 = clock();
  gettimeofday(&start, NULL);

  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      a[i][j] = (i + j) / 2;

  gettimeofday(&end, NULL);
  t2 = clock();
  /* End measurement */

  cpu_time_used = (double)(t2 - t1) / CLOCKS_PER_SEC;
  elapsed_time = (end.tv_sec - start.tv_sec) *
     1000000;
  elapsed_time += (end.tv_usec - start.tv_usec);
  elapsed_time /= 1000000;

  printf("CPU Time Used: %f seconds\n", cpu_time_used)
     ;
  printf("Elapsed (Wall-Clock) Time: %f seconds\n",
     elapsed_time);

  return 0;
}
```

The program was executed on an Ubuntu system using the following commands:

```
gcc -o cpu_time_program cpu_time_program.c -lm
./cpu_time_program
```

And the following times were recorded:

Time Type	Time (seconds)
CPU Time Used	28.314363
Elapsed (Wall-Clock) Time	28.371944

4.2 Analysis of Time Measurements

The CPU time measures the amount of time the processor spent executing the program, which amounted to 28.314363 seconds. This metric reflects the time the CPU was actively working on the program's computations.

The elapsed time, commonly referred to as wall-clock time, was 28.371944 seconds. This encompasses all the time from the start to the end of the program's execution, including the time spent waiting for I/O operations and time slices used by other processes.

The observed CPU time is slightly less than the elapsed time. The close proximity of these two values suggests that the system was largely dedicated to running this program, with minimal interference from other processes or significant I/O wait times.

In scenarios where the system is under heavy load from multiple processes, the CPU time would typically be much less than the elapsed time due to context switching and CPU time allocation to other processes. Conversely, a CPU-bound process with minimal I/O operations, running on an idle system, would exhibit CPU time similar to the elapsed time, as seen in this case.

4.3 Conclusions

The measurement confirms that CPU time is a subset of the elapsed time and is generally less than or equal to the elapsed time. The small difference between the two times in this specific execution indicates that the program is CPU-bound with minimal I/O wait time. The results provide insight into the performance of CPU-intensive applications and underscore the importance of considering both CPU and wall-clock times when evaluating the performance of a program.