Performance Analysis

Overview

The purpose of this assignment is to evaluate how loop modification alters the run-time(s), memory usage, and CPU usage of code. For the purpose of consistency within my analysis, I have constrained my node usage strictly to the “elf” nodes. This ensures my test cases are, at-best, somewhat precise over the different array sizes given to the program.

To begin, I first copied pt0.c, mass\_sbatch.sh, getmem.c, & mytime.c from /homes/dan/625. Instead of manually changing the array size every time I needed to make a change, I modified pt0.c to accept command-line arguments. I had to alter the init\_arrays() & count\_array() method, as well as alter some of the global constants at the top. From here, I copied pt0 two times to have a pt0\_unrolled, and a pt0\_tiled. This helped me keep track of the code I was changing. I also wrote 3 distinct shell scripts which take a variable ‘i’, and run the executable file of pt0, pt0\_unrolled, & pt0\_tiled. I then altered the mass\_sbatch.sh script to contain a loop, which iterates the discrete range of values {8000, 80000, 800000, 8000000}. Each value in the range of the loop was passed into each executable file and then ran on the respective elf-nodes. This allowed me to simply compile each C-file and then run mass\_sbatch.sh to automate most of the work. This submitted 5 jobs of each array case, for all the pt0 files. So, each of my mass batch runs submitted 60 jobs to the elf nodes at a time. I did these three times (for each code alteration) to alleviate any anomalous case which may skew my results. I then used GREP to through each case into a .csv, which I later edited in Microsoft Excel.

Tests & Code Scenario’s: Time & Temporality

To see how my altered code (pt0\_unrolled & pt0\_tiled) ran, I tested each alteration I would make to each files count\_array() method. I came down to two primary test cases or attempts with each optimization technique. With loop unrolling, my initial attempt was to unroll the inner loop and simply repeat the process the loop accomplished by repeating the code. The speed up on this code scenario varied from .0974 ms (array size of 8,000x16), to 98.6899 ms (array size of 8,000,000x16). My second attempt for unrolling gave me slightly better results. In this code scenario, I used the altered code from my first attempt and created a pointer to the “i’th” index of char\_array. The speed up in this scenario was slightly better overall with a .2427 ms (array size of 8,000x16), to 128.5469 ms (array size of 8,000,000x17) speed up. The average time of each case is depicted on **Figure 1** & **Figure 2**:

**Figure 1:** This graph shows the time difference between each loop-optimization technique and the original (pt0). We can see that unrolling the inner loop gave a more efficient run temporally. This is due to caching and locality.

Tests & Code Scenario’s: Virtual & Physical Memory

In my first attempt at tiling, I chose to block the outer loop by 2 and block the inner loop by 4. From here, I repeated the process of accessing the character and adding it to the char\_counts array 4 times. Basing it on runtime alone, this method proved to be inefficient with smaller array sizes. With an array of size 8000x16, my speed up was only .0241 ms. With an array size of 8,000,000x16, I saw observed a 39.6898 ms speed-up. In my next attempt, I tried blocking the outer loop by 4, and the inner loop by 4, and unrolling the actions as before, by repeating the code 4 times and incrementing the column being accessed. This optimization proved to be more efficient than the previous and gave my code speed-ups of .0671 (array size of 8,000x16), to 40.5256 ms (array size of 8,000,000x16). I had one more attempt at tiling, with the outer and inner loop setup the same as the 2nd attempt, but instead of unrolling the actions of the innermost loop and incrementing ‘j’ 3 times, I used 2 more inner loops which iterated through each respective square given by blocking the 2 outter loops. This proved to be slower in some cases than the original pt0. I believe this is due to the Boolean expression required to run loops, as its evaluation would not be local to the data of the array I was trying to access. The relation between runtime and array size for the 1st code scenario tiling is shown above. The 2nd code scenario is shown on **Figure 2** below.

**Figure 2:** The optimized runtimes vs the original for the second code scenario are depicted here. The runtime overall was marginally longer. Loop unrolling yielded a greater yield than tiling once again.

Memory is a significant factor when considering caching and locality. The virtual & physical memory remained relatively the same from on different trials. For instance, when I had an 8000x16 array, when running pt0, pt0\_unrolled, & pt0\_tiled, I found that each of them used about the same amount of memory. The virtual memory for pt0\_unrolled & pt0\_tiled was higher by 4 kilobytes throughout all of my tests. As an example, when pt0 used 316680 kilobytes when given an 8,000,000x16 array, pt0\_unrolled & pt0\_tiled both used 316684 kilobytes. This was true on every test and every different test run. I believe this because I created and allocated memory for a double pointer of characters to be used for storing and passing around in each of the pt0’s, instead of the original 2D array which still existed in the original pt0.c file.

Conclusion

Overall, it appears that unrolling is superior in this case. With tiling, I kept data I wanted to work with in my working set (the cache on the CPU). As a result, I increased the number of cache hits which were successful. This is because the program maintained a good cache locality. I think tiling would have been more effective if our data sets would have been enormous (GB, TB, PB range), because we could utilize more of the CPU cache.