Alex Breault

CIS 625 – Performance Analysis

To begin, we were given the task to evaluate the performance gain/hurt from different techniques. The given code is very simple; it creates an array, assigns random character values throughout the array, and then goes through and counts the number of times that character was used. It prints the results in .out file that has sections like the following image.

```
a 601
b 629
c 662
d 640
e 604
f 568
g 608
h 647
i 641
j 611
k 571
l 642
m 638
n 653
o 624
p 619
q 635
r 576
s 586
t 589
u 610
v 605
w 627
x 598
y 608
z 608

Total characters:  16000
DATA, 1000, 100, elf36, 0.067000
```

The code counts each iteration of a, b, c, etc. and prints how many times that letter was used. The bottom line prints DATA before any values to help narrow down the useful data. The first number on the line represents the array size used. The second number represents the number of tasks used for the job. The next value is 'elf36' which represents the node that exact job was ran on. The last number on the line represents the runtime of that specific job in milliseconds.

The two techniques assigned to evaluate were unrolling and tiling. The current method to count the array was to loop through every value in the array and count each one. When the array size stays small, this method will work perfectly fine. As the array size increases, it can get extremely long and this is where you would use tiling or unrolling. Tiling is an optimization technique used to maximize the number of cache hits. In simple terms, tiling splits all computations into very small *tiles* to allow everything to be done in the cache to increase speed and

productivity. Unrolling is the unwinding of a loop to try and speed things up by keeping computations in the cache and to reduce the number of computations needed. Instead of looping through the array in this example, I hardcoded a counting statement for the size of the array.

To apply the techniques, I changed the count_array method and timed how long that took. For tiling, this is what my method looked like:

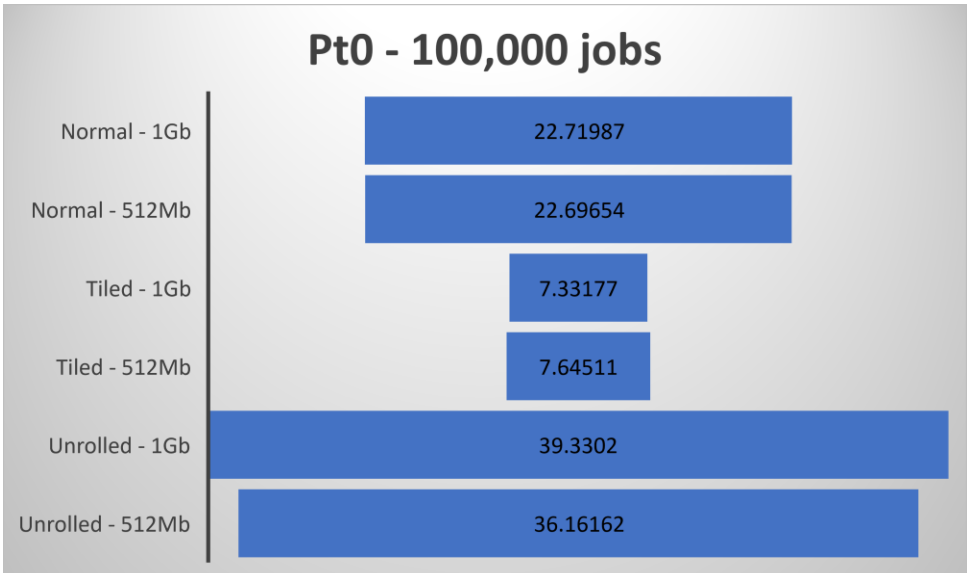
```
void count_array()
{
  char theChar;
  int i, j, x, y, charLoc;

  for ( i = 0; i < array_size; i += 32) {
    for ( j = 0; j < STRING_SIZE; j += 32) {
      for( x = i; x < MIN(i + 32, array_size); x++) {
        for( y = j; y < MIN(j + 32, STRING_SIZE); y++)
        {
          theChar = char_array[x][y];
          charLoc = ((int) theChar) - 97;
          char_counts[charLoc]++;
        }
      }
    }
  }
}
```

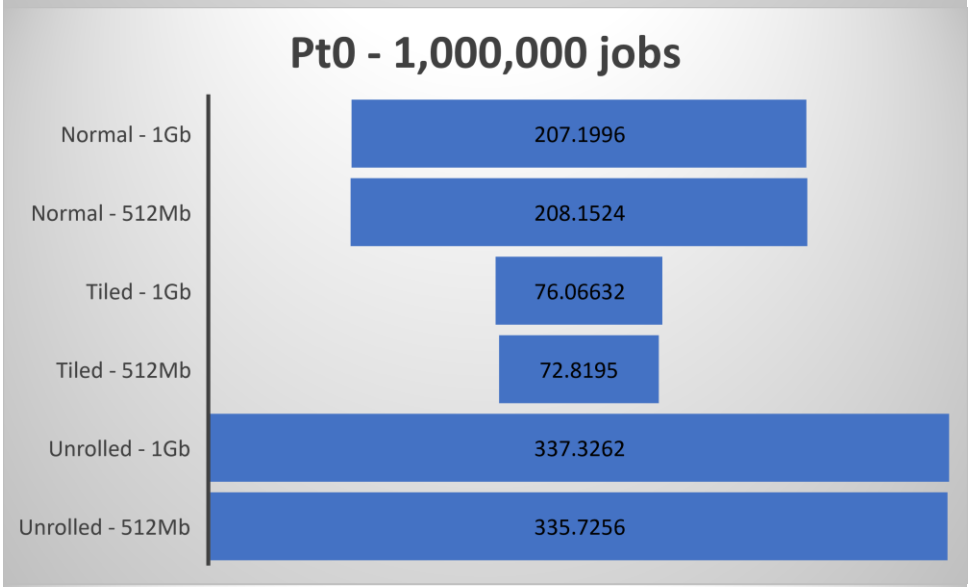
And for unrolling, I unrolled the loops in groups of 8. This loop now looks like this:

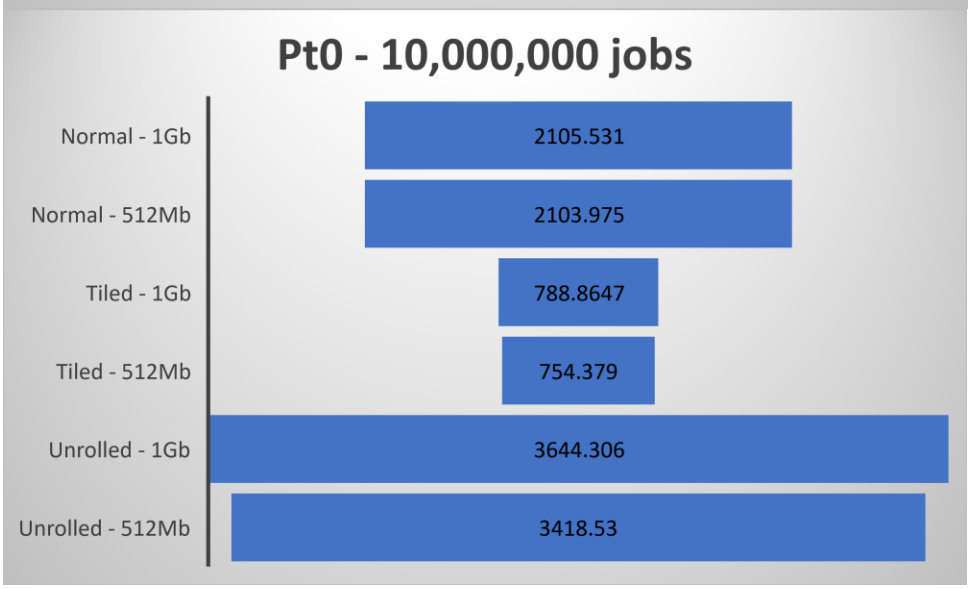
```
void count_array()
{
  char theChar;
  int i, j, charLoc;

  for ( i = 0; i < array_size; i++) {
    for ( j = 0; j < STRING_SIZE; j += 8 ) {
      theChar = char_array[i][j];
      charLoc = ((int) theChar) - 97;
      char_counts[charLoc]++;
      theChar = char_array[i][j+1];
      charLoc = ((int) theChar) - 97;
      char_counts[charLoc]++;
      theChar = char_array[i][j+2];
      charLoc = ((int) theChar) - 97;
      char_counts[charLoc]++;
      theChar = char_array[i][j+3];
      charLoc = ((int) theChar) - 97;
      char_counts[charLoc]++;
      theChar = char_array[i][j+4];
      charLoc = ((int) theChar) - 97;
      char_counts[charLoc]++;
      theChar = char_array[i][j+5];
      charLoc = ((int) theChar) - 97;
      char_counts[charLoc]++;
      theChar = char_array[i][j+6];
      charLoc = ((int) theChar) - 97;
      char_counts[charLoc]++;
      theChar = char_array[i][j+7];
      charLoc = ((int) theChar) - 97;
      char_counts[charLoc]++;
    }
  }
}
```

These both allow for more, if not all, computations to be completed inside of the cache to massively speed up counting the arrays.

For testing, I tested using only elf nodes and submit the jobs via a sbatch script. After testing, I have concluded that tiling made everything run much faster and that unrolling did not help at all as the array size grew. The following graphs show the average run time for each technique using two different memory sizes for the CPU- 512Mb or 1Gb. The runtimes are in milliseconds, but it shows the massive improvement tiling brought.


Pt0 - 100,000 jobs
Normal - 1Gb
22.71987
Normal - 512Mb
22.69654
Tiled - 1Gb
7.33177
Tiled - 512Mb
7.64511
Unrolled - 1Gb
39.3302
Unrolled - 512Mb
36.16162


Pt0 - 1,000,000 jobs
Normal - 1Gb
207.1996
Normal - 512Mb
208.1524
Tiled - 1Gb
76.06632
Tiled - 512Mb
72.8195
Unrolled - 1Gb
337.3262
Unrolled - 512Mb
335.7256


Pt0 - 10,000,000 jobs
Normal - 1Gb
2105.531
Normal - 512Mb
2103.975
Tiled - 1Gb
788.8647
Tiled - 512Mb
754.379
Unrolled - 1Gb
3644.306
Unrolled - 512Mb
3418.53

These graphs show that, as the array size and number of jobs increase, the runtimes grow but tiling gives us the speedup that we desired. Because tiling was the only technique that showed some sort of speedup, this means that most of the computations, if not all, were done within the cache. The cache can do small computations extremely quickly but is limited on the size per level. For tiling, the computation power averaged 11,310 jobs per millisecond (number of jobs divided by the average runtime) compared to 4,567 for the normal method and 2,710 for unrolling. This show the true power and speed the processor's cache holds.

Also while running, I noticed that giving the processor only 512 megabytes of memory versus 1 gigabyte helped speed things up. When running all the tests, the numbers for 512 megabytes of memory were, on average, 3 to 6 percent lower than when running with 1 gigabyte of memory. I think the numbers are different because the processor has more memory available to it, it decides to move some computations to memory rather than in the cache since it's available to it. When moving computations to memory, this is always going to be slower than doing them in the cache.

Overall, I saw moderate improvement with tiling but my results for unrolling are inconclusive. I thought unrolling would lead to the same speedup as tiling but this was not the case. My algorithms were correct (hopefully) so therefore the program was just not optimized to be unrolled.