Bryan Acosta's Hardware HW 2 writeup using LaTeX

Eids: ba25389

11/23/2021

1 Calculating Pi:

The algorithm works by first squaring each element of the array, then adding all of the x and y elements to each other to result with one long array. Then, it adds all of the elements that are less than 1 into a single number. Then, it is multiplied by 4 and divided by the total size of the original array - and that number is equal to pi.

2 Original Code:

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

using namespace std;

int main (){
  srand (time(NULL));
  int size = 100000;
  double size_in_double = 100000;
  double sumnum = 0;
  int count = 0;
  struct timeval start, end;
  float delta;

  vector<double> xvalues(size);
  vector<double> yvalues(size);

  for (int i = 0; i < size; i++) { // step 1: assigns random value to each element
    xvalues[i] = double(rand())/double(RAND_MAX);
    yvalues[i] = double(rand())/double(RAND_MAX);
  }
  // gettimeofday(&start, NULL);

  vector<double> xvalues_squared(size);
  vector<double> yvalues_squared(size);

  for (int i = 0; i < size; i++) { // squares each element
    xvalues_squared[i] = xvalues[i] * xvalues[i];
    yvalues_squared[i] = yvalues[i] * yvalues[i];
  }

  vector<double> added_values(size);

  for (int i = 0; i < size; i++) { // add x and y
    added_values[i] = xvalues_squared[i] +  yvalues_squared[i];
  }


  for (int i = 0; i < size; i++) { // sums all values <= 1
```

```
    if (added_values[i] <= 1) {
    sumnum += 1;
    }
  }
  //cout << sumnum << endl;

  double final_value = (4*sumnum) /size_in_double;

  //cout << final_value << endl;

  gettimeofday(&end, NULL);
  delta = ((end.tv_sec-start.tv_sec)*1000000u + end.tv_usec-start.tv_usec)/\
    1.e6;
  double memorie = (64*8*size)/(1024*1024);
  cout << "Precision: Double"  << endl;
  cout << "Sample Size: 10000" << endl;
  cout << "Total Memory footprint (MB): " << memorie << "MB"  << endl;
  cout << "Total Memory footprint (GB): " <<  memorie/1025 << " GB" << endl;
  cout << " " << endl;
  cout << "Pi: " << final_value << endl;
  cout << "time: " << delta << endl;
    return 0;
}
```

3 Assignment part 1:

This is the output when ran:

```
Precision: Double
Sample Size: 10000
Total Memory footprint (MB): 48MB
Total Memory footprint (GB): 0.0468293 GB

Pi: 3.142
time: 8.02054e+12
```

4 Assignment part 2:

Explain how often data is moved from memory into the CPU (and back!) for the individual steps. Does the cache (or the caches) provide any help to boost performance? ans. the data is accessed, moved, and edited each time that the computer runs through the loop and each time that is accesses the vector. the cache makes the needed data easier and faster to access. Why would the calculation give an incorrect result for a large sample size (like the one we are using here, see 'Part 4'), if the variable 'num inside' was a single precision number?

ans. The memory allocated for a single precision number was not large enough to contain the number. Also, with a large enough number, the random function can start malfunctioning and stop being random.

5 Assignment part 3:

old code:

```
vector<double> xvalues_squared(size);
  vector<double> yvalues_squared(size);

  for (int i = 0; i < size; i++) { // squares each element
    xvalues_squared[i] = xvalues[i] * xvalues[i];
    yvalues_squared[i] = yvalues[i] * yvalues[i];
  }

  vector<double> added_values(size);

  for (int i = 0; i < size; i++) { // add x and y
    added_values[i] = xvalues_squared[i] +  yvalues_squared[i];
  }


  for (int i = 0; i < size; i++) { // sums all values <= 1
    if (added_values[i] <= 1) {
      sumnum += 1;
    }
  }
  double final_value = (4*sumnum) /size_in_double;
```

is changed into:

```
  double x,y,z;
  for (int i = 0; i < size; i++) { // squares each element
    x = xvalues[i];
    x = x*x;
    y = yvalues[i];
    y = y*y;
    sum = x + y;
    if (sum <= 1) {
    totalsum += 1;
    }
  }
  double final_value = (4*totalsum) /size_in_double;
```

and the output changes from:

```
Precision: Double
Sample Size: 10000
Total Memory footprint (MB): 48MB
Total Memory footprint (GB): 0.0468293 GB

Pi: 3.14624
time: 0.005875
```

into:

```
Precision: Double
Sample Size: 10000
Total Memory footprint (MB): 48MB
Total Memory footprint (GB): 0.0468293 GB

Pi: 3.14304
time: 0.002748
```

6 Assignment part 4:

The numactl command allows you to compile/run a program on a specific node (useful for idev. The two flags specify with node to use, specifically the one the idev is running on.

7 Assignment part 5:

The execution speed is faster for the optimized code because it no longer has to access the vectors one by one. Now it only calls an element from a vector once and assigns it to a variable, making it a lot easier and faster to access. Also, now it doesn't run through as many loops, so it doesn't have to access the same memory as many times.