Da Lu

Hw1

Parallel Programming

2/27/14

dlu6@jhu.edu

Loop efficiency

1 a)

b)

c) SerialDataFirst is more efficient because of the cache situation. When running DataFirst, the program loops through the data array only once and it runs the filter array data\_length times. Since the filter array is way smaller compared to data, it can be stroed in the cache first time we read it. They can be easily accessed later from the cache in subsequent reads. When running FilterFirst, the program loops through the filter array once and the data array filter\_length times. The data, however, is large and may not be stored in the cache. This causes the program to read from memory each time through the data array, which causes it to run slower. This also explains why as the filter length increases, the FilterFirst run becomes increasingly slower than the DataFirst run.

d) Relative performance measures normalized runtime/filter size. The relative performance would not be affected by changing filter size. Because, shown in problem 1a, as the filter size increases by a factor by 2, the runtime also increases by a factor of 2. Since we calculate relative performance by data\_length \* filter\_length / runtime, increase in filter\_length and increase in runtime cancel out, leaving the relative performance to be independent of filter size.

Loop Parallelism

1a)

b) For both parallel filter first and parallel data first speedup, they both ran fastest when number of threads is set to 4. By looking at their performance plots, it is realized that they are sublinear speedup, because operation per second did not increase at a linear pace. It started falling off after number of threads is equal to 4, because it has reached its maximum improvement when number of threads is equal to 4. We ran this program on Amazon High-CPU 8core instances (c3.2 large), it should increase our performances significantly because it runs faster than normal servers.

c)

d)

For both parallel filter first and parallel data first scaleup, they had the best performance when both data size and number of threads are both 2. By looking at their performance plots, it is fairly obvious that they are sublinear scaleup. , because scaleup did not have a slope of 1. Instead it goes down as problem size increases. It started falling off after problem size is increased by 2. It started falling off because of startup costs, interference and skew. We ran this program on Amazon High-CPU 8core instances (c3.2 large). However, it should not affect our performances since we ran all instances on the same machine anyways.

2 Parallel Performance:

a)

Parallel data first was faster. The reasoning behind it is similar to that of 1 c).

This is because of the cache situation. . When running DataFirst, the program loops through the data array only once and it runs the filter array data\_length times. Since the filter array is way smaller compared to data, it can be stroed in the cache first time we read it. They can be easily accessed later from the cache in subsequent reads. When running FilterFirst, the program loops through the filter array once and the data array filter\_length times. The data, however, is large and may not be stored in the cache. This causes the program to read from memory each time through the data array, which causes it to run slower. This also explains why as the filter length increases, the FilterFirst run becomes increasingly slower than the DataFirst run.

b)

Our best performance occurred when we used 4 as number of threads, and our overall speedup is about 3.917 times. We apply the formula 1 / (1-P) + P/s = speedup, s = 4, speedup = 3.917. 1 / (1-P) + P/4 = 3.917, P = 99.3%. It means that 99.3% of the computation maybe the subject of speedup.

c)

There are 3 factors against ideal speedup or scaleup: Startup costs, interference, and skew. For our program, startup costs maybe the cost to locate our data array and filter array from our memory. We need to find these two arrays in memory before I do anything. Interference also slowed down our performance. We need to repeatedly access memory and cache. Skew may also slows down our performance, it happens when not all tasks finish at the same time.

An Optimized Version

1 a)

My best optimization is to use loop unrolling. Loop overheads are relatively high when each iteration has a small number of operations. So we have two loops in filter.c, one is the inner loop; the other is the outer loop. I changed the inner loop so that it increments by 2 every iteration. That way each iteration will have twice as many operations as before, and loop overheads become lower. They are pretty effective as I compared results using loop unrolling with the old results. The runtime improves approximately 33%. Here is what my code looks like:

for (int y=0; y<filter\_len; y++) {

/\* for all elements in the data \*/

for (int x=0; x<data\_len; x+=2) {

/\* it the data element matches the filter \*/

if (input\_array[x] == filter\_list[y]) {

/\* include it in the output \*/

output\_array[x] = input\_array[x];

}

if (input\_array[x+1] == filter\_list[y]) {

/\* include it in the output \*/

output\_array[x+1] = input\_array[x];

}

}

}

b)

I tried loop fission. Loop fission can be applied to loops with poor cache utilization and bad memory access. This optimization could not really work for our program, because our two loops have good cache utilization. Our loops access arrays row by row, so we really can not use loop fission.