CS484 parallel programming homework 1

1 MESI protocol

1.1 start with empty cache line

	PE 0's state	PE 1's state	PE 2's state
Initial state	I	I	I
PE2 reads K	I	I	E
PE2 writes K	I	I	M
PE0 reads K	S	I	S
PE1 reads K	S	S	S
PE1 writes K	I	M	I

1.2 starting with PE 0's cache containing with a copy of K

	PE 0's state	PE 1's state	PE 2's state
Initial state	E	I	I
PE2 reads K	S	I	S
PE2 writes K	I	I	M
PE0 reads K	S	I	S
PE1 reads K	S	S	S
PE1 writes K	I	M	I

2 Cache

2.1

4 words per lines → 16 bytes per lines.
The total size of the cache is: 4096 * 16 * 8 = 512k bytes (*8 because of 8-way associative).

2.2

If we interchange I and j, we will get a miss every two search from cache. Like, first time A[0][0] will be cold miss and A[0][1] will be hit. Because it only bring two elements into the cache so the third time A[0][2] will be miss and A[0][3] will be hit. So it goes on and on, every two search will get a miss, the total times of miss equal 1024K / 2 = 512 * 1024 = 524288
I

2.3

n this kind of code, when j=0 for the first loop, k times of cache miss will occurred. When j=1, 0 time of cache miss will happened. So we find out that pair of j(0,1) will create K times of miss and so will the pair (2,3) (4,5) ….and so on. So the total cache miss will be k * 512 = 512 * 1024 = 524288 times of cache miss.
Compared with the previous one, we found out that the cache miss is same, so we cannot improve the performance by interchange i and j.

3 Loop optimization

3.1
I can found some bottle here, first of all we found it that the two loop in the foo() are work separately. But they can fuse into one loop and use the cache locality of some data that has been brought into the cache which can save a lot of time. Second, It can also make two loop into one loop which save more time of it.

3.2
For the first bottleneck we may rewrite the code that we can fuse the two loop together so that it can use the cache locality of b[] and c[] that brought into cache which can save a   lot of times. More detail is: we may change the loop to :

```
void foo ( double _ a , double _ b , double _ c ) {
i n t i ;
f o r ( i =32; i <(N/ s i z e o f ( double ) ) ; i = i + L) {    // fuse into one loop.
    a [ i-1 ] = b [ i-1 ] * c [ i-1 ] ;
    a [ i ] = b [ i ] + c [ i ] ;          // cache locality here.
}
a[0] = b[0] + c[0]                   // two single part we need to do it outside the loop.
a[511] = b[511] * c[511]
```

4 OpenMP

4.1
The data dependency is that the myIndex may face race condition which lead to the un expected result. More explicitly is that myIndex = A[i] and
B[i] = D[myIndex]+C[muinex] will have problems.

We can change the openMP code as follow to solve the problems.

```
#pragma omp parallel for private(myIndex)
for ( int i = 0 ; i < n ; i++) {
        myIndex = A[ i ] ;
        i f ( myIndex >= 0 && myIndex < n ) {
                B[ i ] = D[ myIndex ] + C[ myIndex ] ;
        }
}
```

4.2

```
int minVal = A[ 0 ] ;
#pragma omp parallel for
for ( int i = 0 ; i < n ; i++) {
#pragma omp critical
        if (A[i] < minVal) {
                minVal = A[ i ] ;
        }
}
```

4.3

NO, the code cannot be parallelized because the code A[i][j]=A[i −1][j]∗B[i]∗C[j] ; always need to wait the previous result to be completed, so it cannot be parallelized.

4.4

4.4.1

The program may encounter load imbalance problem because calculate the sigle multiply and calculate the Fibonacci has completely different time. We may need to dynamically assign the work to the thread instead of static.

4.4.2

```
int compute ( int input , int index )
{
        if ( inde x < N/2 )
        return input ∗ inde x ;
        else
        return Fibonacci( inde x ) ;
}
#pragma omp parallel for schedule (dynamic)
for ( int i = 0 ; i < N; i++)
{
        B[ i ] = compute (A[ i ] , i ) ;
}
```

4.5

4.5.1

No, we cannot parallelized the code with only “pragma omp parallel for”. Because it may cause race condition. The “m -- “ should be critical to all the threads.

4.5.2

Because when the iteration is not fixed we cannot do the parallel. So what I done is to calculate the iteration and then do the parallel.

```
int count=0;
int temp=m;
For (int j=0;j<length_of_B;j++) {
    if (m==0)
        break;
    else {
        if( f(b[i])==0 )
            count++;
        else
            m--
    }
}
Pragma omp parallel for
for ( int i = 0 ; i < count+temp; i++) {
    A[ i ] = f (B[ i ] ) ;
}
```