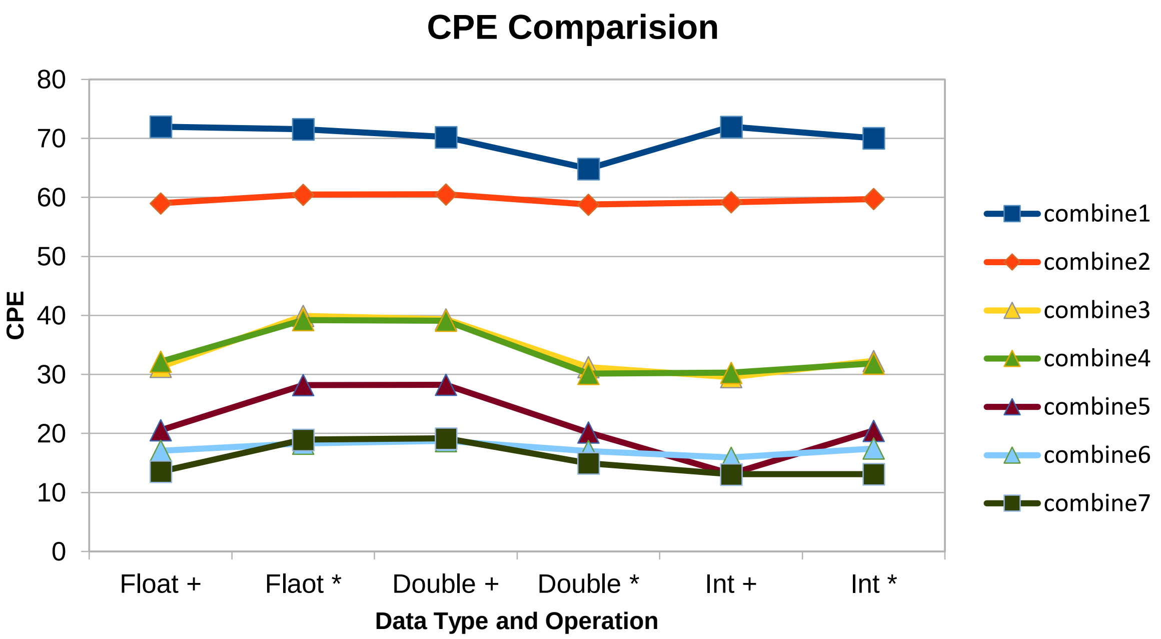
Part 1

a)



As we can see, the results obtained are almost in agreement with the results given in the book. The CPE is observed to go down with every optimization from combine1 to combine7.

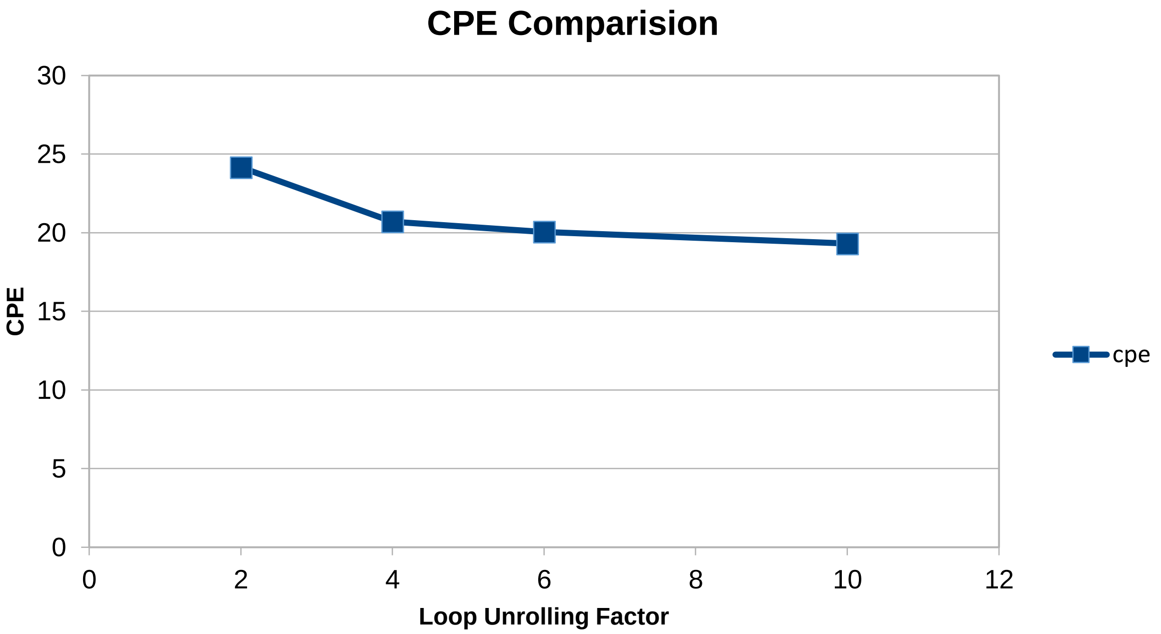
I observed a few random spikes in otherwise linear graphs for the data I obtained for various combinations. Those were probably due the CPU being occupied by some background programs and hence causing a cache miss. In 95% cases, the results obtained were as expected.

The CPE doesn't seem to vary a lot with respect to data types since the vector size is very small. If run for a larger data size, float and double take significantly more time than int. Also, multiplication of larger vectors take longer than the addition.

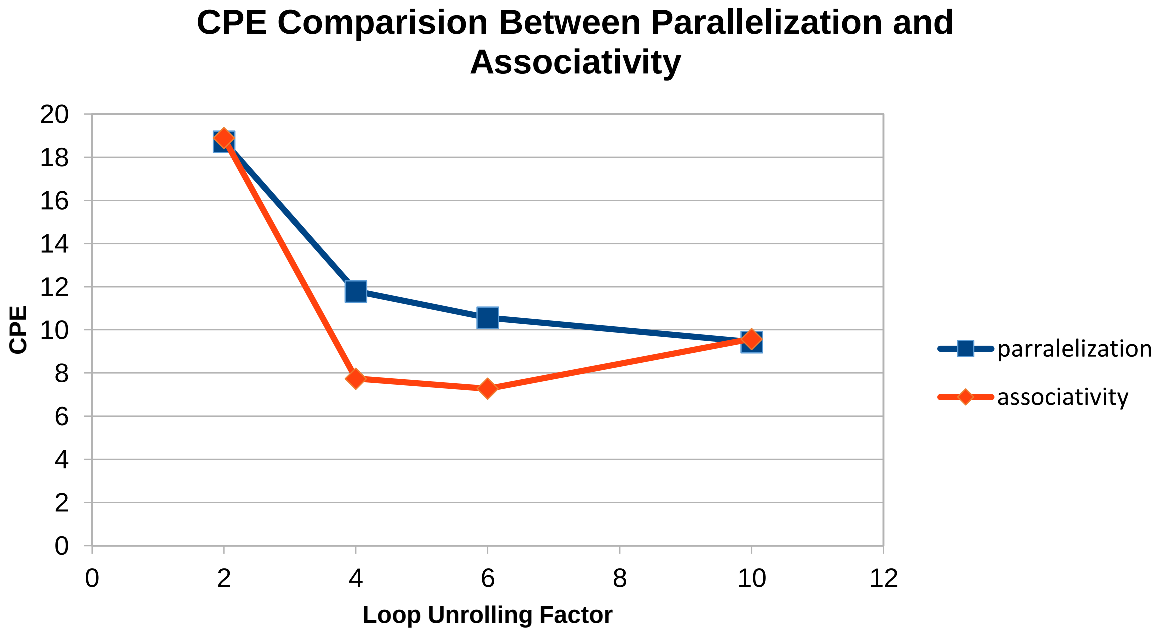
b) I noticed that for factors greater than 6, loop unrolling didn't worsen the performance although it got a little stagnated. On increasing the unrolling factor beyond 6, not a significant increase in the performance is observed but it does improve slightly.

The possible explanation for this could be the increase in number of operations carried out in one single iteration. Since multiply operation takes a signficant number of cycles, increasing the unrolling factor beyond 6 would mean to perform more than 6 multiplications in one iteration. This will also increase the number of registers required to store the intermediate results. All these reasons lead to the performance becoming stagnant beyond a certain unrolling factor.

However, for small unrolling factor, the performance does show a significant improvement and hence proves the benefit of loop unrolling.



c)



Here, we observe that for smaller loop unrolling factors associativity performs better than parallelization. However, for unrolling factors greater than 6, the number of operations per iteration increases and hence we observe an increase in CPE for associativity. Parallelization on the other hand does not take a major hit in performance but is stagnant which is because of the two accumulators to store the result.

Part 2

The program is based on the function in test_combine.c in particular to combine4. In addition to the dot product in combine4, two other optimized methods has been performed which are loop unrolling and parallelization. From the table we can conclude that Loop unrolling has the best performance as the number of cycles in it is least. The second table is for the CPE for various methods performed. Loop unrolling has the least CPE which is 17.7590977444.

| Number of Elements | Dot Product | Dot Product Unrolling | Dot Product Parallelization |
|---|---|---|---|
| 10 | 4941 | 2789 | 3166 |
| 20 | 2720 | 2850 | 2615 |
| 30 | 2992 | 2647 | 2720 |
| 40 | 3271 | 2981 | 2923 |
| 50 | 3810 | 2992 | 3123 |
| 60 | 3926 | 3184 | 3358 |
| 70 | 4260 | 3311 | 3564 |
| 80 | 4535 | 3561 | 3778 |
| 90 | 4898 | 3778 | 3955 |
| 100 | 5217 | 3923 | 4141 |
| 110 | 5510 | 4129 | 4347 |
| 120 | 5884 | 4347 | 4593 |
| 130 | 6194 | 4622 | 4782 |
| 140 | 6556 | 4709 | 4941 |
| 150 | 6832 | 4869 | 5463 |
| 160 | 7180 | 5014 | 5423 |
| 170 | 7528 | 5292 | 5626 |
| 180 | 7835 | 5553 | 5814 |
| 190 | 8386 | 5698 | 6061 |
| 200 | 8488 | 5933 | 6278 |

|  | Dot Product | Dot Product Unrolling | Dot Product Parallelization |
|---|---|---|---|
| CPE | 28.7626315789 | 17.7590977444 | 19.5701503759 |

There are two optimization methods in the code:

- Loop Unrolling –

  It is a method in which the number of iterations is reduced as compared to the normal dot product method. It reduces the number of iterations by increasing the number of elements computations in each iteration. Loop unrolling performed in the code computes the dot product of two sets of numbers at a time and adds it to the accumulator. Unrolling factor used is 2. The remaining elements which are left at the end of the vectors are computed by the second loop one at a time.

  It improves the performance in two ways:-

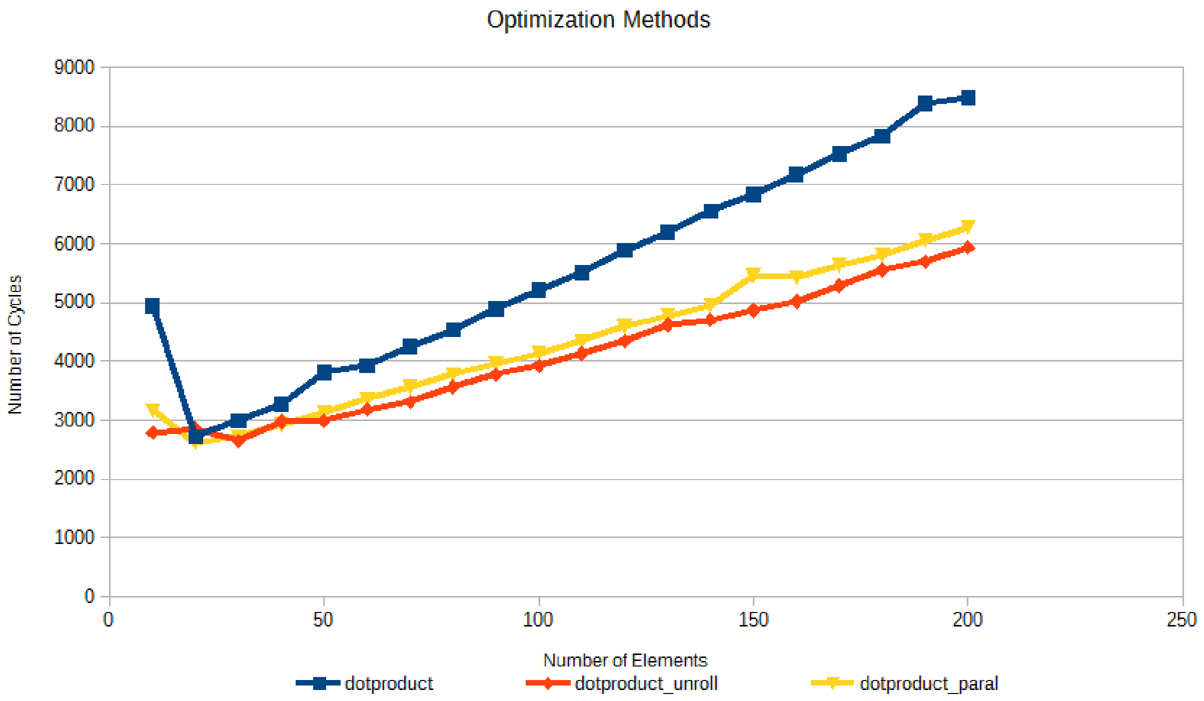
  1. Firstly it reduces the number of operations like loop indexing and conditional branch which do not directly contribute to the program result.
  2. It also exposes ways in which to reduce the number of computations in the critical paths.

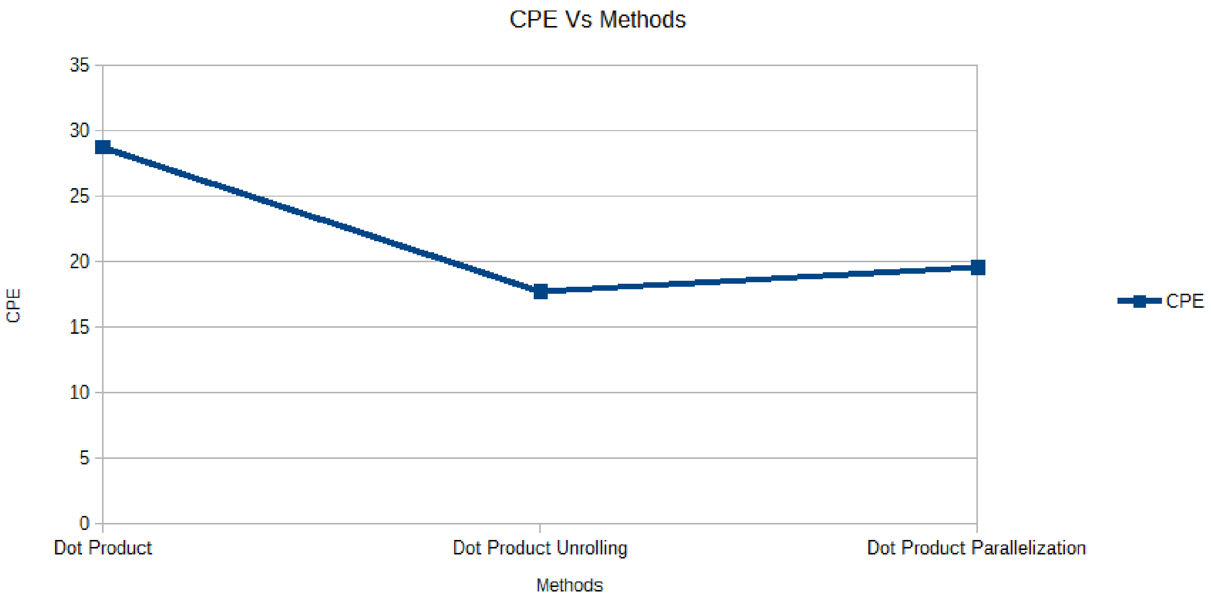
  In the code the first loop performs in such a way that the loop index i is incremented by 2 in each iteration. In each iteration the dot product is applied to elements i and i+1 of both vectors. As index is incremented by 2 every time it is not possible that the length of the array will be a multiple of 2. Firstly care should be taken that loop should not over run the vector bounds. This is done by setting loop limit to be n-1 (n = length of the vector). Secondly the remaining elements need to be computed thus for that another loop is introduced which computes the dot product of the remaining elements.
  This is the best optimization technique out of the two used.

- Parallelization –

  As seen in loop unrolling the functional units are fully pipelined which are performing addition and multiplication. These units are performing operation at every new cycle but the code cannot take advantage of this as the result is accumulated in a single variable. Thus to overcome this and increase performance in our code we have split up the combining set of operations into two parts and combined the results in the end.

  In the code we have used acc0 and acc1 to perform the operations. So at a particular index value of the loop dot product of two set of values are computed and stored in acc0 and acc1. Then another loop is used to compute results of remaining elements. Finally both acc0 and acc1 are combined.


Optimization Methods
Number of Cycles
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0
50
100
150
200
250
Number of Elements
dotproduct
dotproduct_unroll
dotproduct_paral


CPE Vs Methods
CPE
35
30
25
20
15
10
5
0
Dot Product
Dot Product Unrolling
Dot Product Parallelization
Methods
CPE

Part 3

For the init_vector1, I used the fRand function provided in the code to generate a random data set. For init_vector2, I simply assigned the value of loop variable i to generate a linear data set.

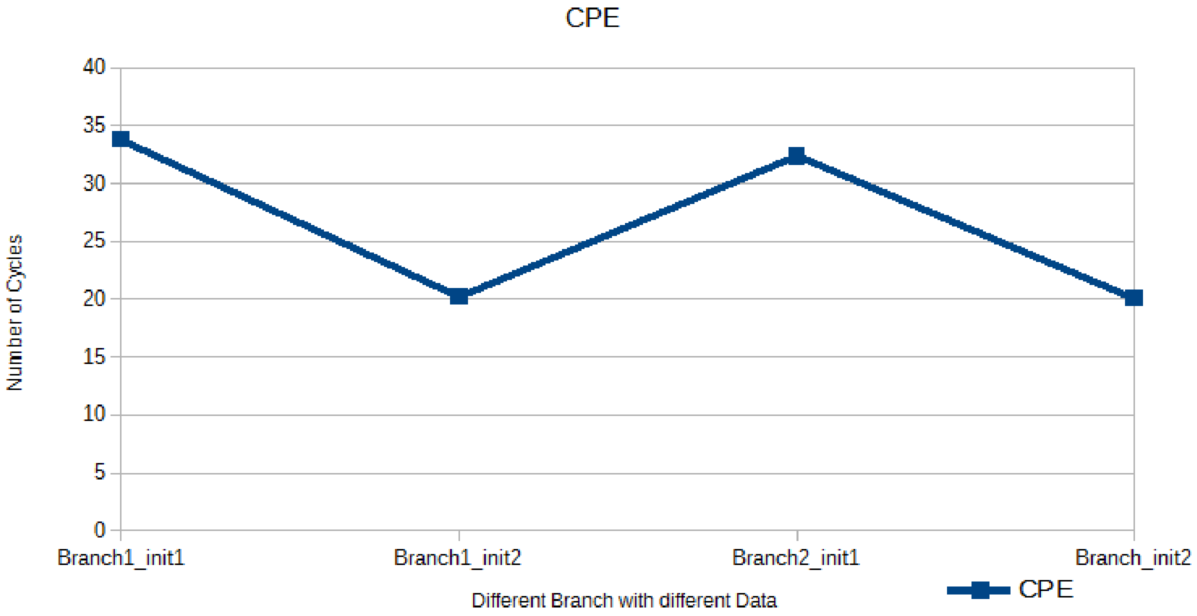
The code was as follows:

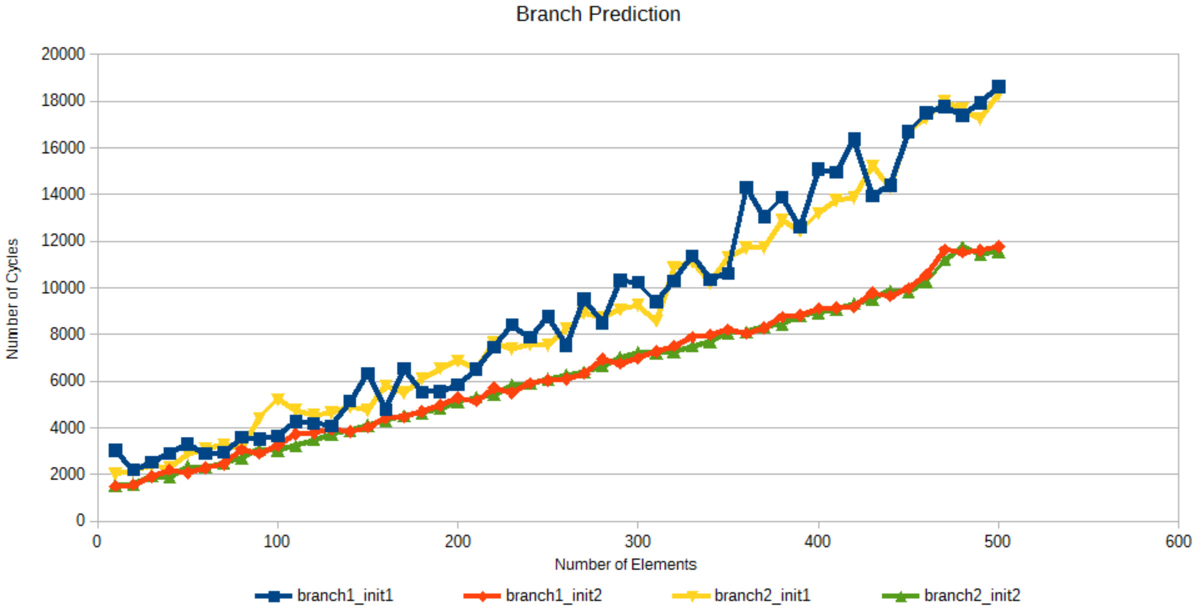
```
/* initialize vector with first pattern */
int init_vector1(vec_ptr v, long int len)
{
        long int i;
        double fRand(double fMin, double fMax);
        if (len > 0) {
          v->len = len;
          for (i = 0; i < len; i++)
                v->data[i] = fRand(100.0,5000.0);
          return 1;
        }
  else return 0;
}

/* initialize vector with another */
int init_vector2(vec_ptr v, long int len)
{
        long int i;
        double fRand(double fMin, double fMax);

        if (len > 0) {
          v->len = len;
          for (i = 0; i < len; i++)
                v->data[i] = i;
          return 1;
        }
  else return 0;
 }
```

| | branch1_init1 | branch1_init2 | branch2_init1 | branch_init2 |
|---|---|---|---|---|
| **CPE** | 33.8280528211 | 20.2414597839 | 32.3641248499 | 20.101757503 |


CPE
Number of Cycles
40
35
30
25
20
15
10
5
0
Branch1_init1
Branch1_init2
Branch2_init1
Branch_init2
Different Branch with different Data
CPE


Branch Prediction
Number of Cycles
20000
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
0
100
200
300
400
500
600
Number of Elements
branch1_init1
branch1_init2
branch2_init1
branch2_init2

Part 4

For this part I tried the following methods:

- The normal method mentioned in the book.
- Modified it by introducing loop unrolling.
- Used Accumulators for parallelism.
- Used Associativity for parallelism.

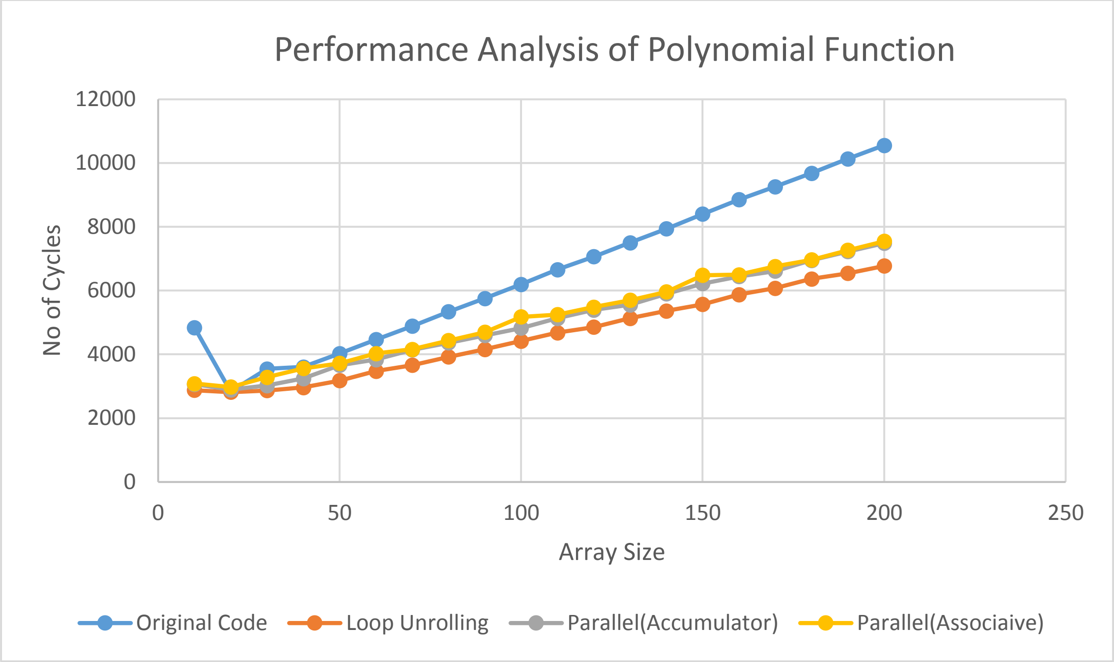
The problem statement here is to calculate a polynomial calculation for a given value of “x”. For this the coefficients of the equation were given by an array. I used an abstract array in which we can have array of any data type and then calculate the expression. So I modified the code such that the loop will run “n” times where n is the length of the array. So as an array is involved in the calculation that means looping is required, therefore there are a lot of optimizations that can be tried on it.

1. Normal Method:
   This method was the one given in the book, I modified the whole code according to our abstract data structure and implemented it.

2. Loop Unrolling:
   I applied loop unrolling to the code with an unrolling factor of 2. This improved the performance by a great factor and reduced the number of cycles by almost half for large array sizes. In this method a single loop executed the expression for two consecutive array elements, thereby decreasing the number of loops required.

3. Accumulators for parallelism:
   In this method I modified the code such that same computation is done for half of the array and also for the other half but using half the looping. This method also decreased the time as compared to the original method.

4. Associativity for parallelism:
   This method is similar to loop unrolling. The only difference is the order in which the operations are performed. In this the associative operation is operated first and then added. Loop unrolling has the opposite.



From the above graph I conclude that the best performance is given when using Loop Unrolling technique and it is clearly evident that all the modifications of original code are much better than the original code.

Both the parallel codes gave almost similar performance and it can be seen that there is not much difference in loop unrolling and parallel codes.

After seeing the assembly level code for each optimization I concluded that the number of operations inside the loop are the least in the case of loop unrolling. I also tried loop unrolling for various unrolling factor but the best performance was given by the unrolling factor of 2.

**CPE Values for all the functions:**

- *Original: 33.9954*
- *Loop Unrolling: 22.49*
- *Parallel (Accumulator): 25.08*
- *Parallel (Associative): 24.68*

Part 5

1. 2 days.
2. No
3. No
4. No