Name: Chen-yang Yu
Student ID: 862052273

Part 1: Modify the parallel Sieve of Eratosthenes program in class so that the program does NOT set aside memory for even integers.

Since we know basically all prime number is odd number, we allocate only half of size memory for each process. As the original algorithm, we pick a prime less than square root(starting from 3) of input number n (prime*prime <= n) and mark the multiples of this prime. But in this approach we only mark odd number and accumulate odd number in the last. Notice that our algorithm will miss 2, which is the only even prime number. Add one to counter for prime 2 in the final result.

```
for (i = first; i < size; i += prime){
        //if(id == 1)
            //printf("\tmark:%d\n",i+low_value);
        tmp = i+low_value;
        if((tmp)%2==0){
            //printf("\tits an even!!\n");
        }else{
            marked[(tmp-low_value)/2]=1;
        }
}
```

The process zero (whose id is 0) response for broadcasting the next prime to the rest of process. Since now we only allocate half of array for each process, each element i in marked[] represents the odd number i*2 + 3

```
  if (!id) {
     while (marked[++index]);
     //prime = index + 2; //part1
     prime = (index*2) + 3;
     //printf("\tbroadcast prime:%lld\n",prime);
  }
  MPI_Bcast (&prime,  1, MPI_INT, 0, MPI_COMM_WORLD);
```

Part 2: Modify the parallel Sieve of Eratosthenes program in Part 1 so that each process of the program finds its own sieving primes via local computations instead of broadcasts.

Each process maintain a marked_prime[] to calculate the next prime instead of waiting for process zero broadcast. Process zero still responses for accumulating global_count in the last.

```
long long int first_0 = prime*prime - low_value_0;
for(i = first_0; i < (sqrt(n)); i += prime){
        tmp = i+low_value_0;
        if((tmp)%2==0){
            //printf("\tits an even!!\n");
            }else{
                marked_prime[(tmp-low_value_0)/2]=1;
            }
}
0
  if (!id) {
     while (marked[++index]);
     //prime = index + 2; //part1
     prime = (index*2) + 3;
     printf("\tbroadcast prime:%lld\n",prime);
  }
  MPI_Bcast (&prime,  1, MPI_INT, 0, MPI_COMM_WORLD);
if
  if(1){
     while(marked_prime[++index]);
     prime = (index*2) + 3;
  }
```

Part 3: Modify the parallel Sieve of Eratosthenes program in Part.2 so that the program can have a more effective use of caches.

First we notice that the marked_prime[] array maintained by each process is reused when we check every prime number. It would be more efficiency if we can access elements of marked_prime[] through cache reuse. This may happen when we access an element of marked_prime[], a certain block of marked_prime[] will be also in cache. We use 14,000 as a block (this is the result of trial, the result is approximately the same when we choose block as 10,000~20,000). And check multiple prime in this block(4 in our case).

```
        for(i = 0; i< size; i+=B){
        //while(i<=size){
            for (j = first; j < MIN(size,i+B); j += r){
                tmp = j+low_value;
                    if((tmp)%2==0){
                    }else{
                        marked[(tmp-low_value)/2]=1;
                    }
            }
            first = j;
            for (j = first2; j < MIN(size,i+B); j += r2){
                tmp = j+low_value;
                    if((tmp)%2==0){
                    }else{
                        marked[(tmp-low_value)/2]=1;
                    }
            }
            first2 = j;
            for (j = first3; j < MIN(size,i+B); j += r3){
                tmp = j+low_value;
                    if((tmp)%2==0){
                    }else{
                        marked[(tmp-low_value)/2]=1;
                    }
            }
            first3 = j;
            for (j = first4; j < MIN(size,i+B); j += r4){
                tmp = j+low_value;
                    if((tmp)%2==0){
                    }else{
                        marked[(tmp-low_value)/2]=1;
                    }
            }
            first4 = j;
```

Fig.Left: find multiple of each prime in block

```
            long long int first_0 = r*r - low_value_0;
            for(i = first_0; i < (sqrt(n)); i += r){
                tmp = i+low_value_0;
                if((tmp)%2==0){
                    //printf("\tits an even!!\n");
                }else{
                    marked_prime[(tmp-low_value_0)/2]=1;
                }
            }
    if(!Flag_prime){
            first_0 = r2*r2 - low_value_0;
            for(i = first_0; i < (sqrt(n)); i += r2){
                tmp = i+low_value_0;
                if((tmp)%2==0){
                    //printf("\tits an even!!\n");
                }else{
                    marked_prime[(tmp-low_value_0)/2]=1;
                }
            }
            first_0 = r3*r3 - low_value_0;
            for(i = first_0; i < (sqrt(n)); i += r3){
                tmp = i+low_value_0;
                if((tmp)%2==0){
                    //printf("\tits an even!!\n");
                }else{
                    marked_prime[(tmp-low_value_0)/2]=1;
                }
            }
            first_0 = r4*r4 - low_value_0;
            for(i = first_0; i < (sqrt(n)); i += r4){
                tmp = i+low_value_0;
                if((tmp)%2==0){
                    //printf("\tits an even!!\n");
                }else{
                    marked_prime[(tmp-low_value_0)/2]=1;
                }
            }
```

Fig.Right: each process maintain marker_prime[]

```
            while(marked_prime[++index] && index<sqrt(n));
            r = (index*2) + 3;
if(r>=1000000)
        Flag_prime = 0;
if(!Flag_prime){
            while(marked_prime[++index] && index<sqrt(n));
            r2 = (index*2) + 3;
            if(r2*r2>n)
                r2 = r;
            while(marked_prime[++index] && index<sqrt(n) );
            r3 = (index*2) + 3;
            if(r3*r3>n)
                r3 = r;
            while(marked_prime[++index] && index<sqrt(n));
            r4 = (index*2) + 3;
            tmp = r4;
            if(r4*r4>n)
                r4 = r;
            //printf("p1 = %lld, p2 = %lld, p3 = %lld, p4 = %lld\n", prime, prime2, prime3, prime4);
}
```

Fig: use marker_prime[] to pick 4 prime for next iteration

We check part_1, part_2, and part_3 result in one node(32 process) to eight nodes(256 process) separately. The result is as following.