Parallel Computing Assignment (MPI)

1. For the sieve problem, the challenge was to reach the 10^9 mark. The system would behave erratically after 10^8 running sometimes out of memory and failing. So, splitting the data into chunks of 10^7 if the upper bound (N) is over 10^7. Then for each chunk the program processes and compute the max primes while keeping a count of the primes found. I tried two techniques – one where it uses the given template to communicate primes (until sqrt(N)) through the first chunk and store the required primes as they arrive with my local processes. Then it finds primes for the rest of the chunks and computed the mean and rms. The other approach I tried gave me better results (tested until 10^{10}). In the submitted method, the program finds primes in one processor until sqrt(N) (I tried multi-processor, but it's much slower since sqrt(N) is usually a very small number w.r.t. N for the values we use). Then it broadcasted the primes to all processes in a batch and chunk-wise found the Max and count, performing reduction on each at the end.

   Results:

Processors	1	2	4	8	16	32	64
Time (sec)	5.384	3.564	3.141	2.064	0.879	0.265	0.121

   # of primes less than 10^9 = 50847534

   Maximum prime less than 10^9 = 999999937

2. For the FDTD problem, I used data generator from the *fdtd-seq.c* and wrote the data into the file to be used as in the *floyd.c* as an input to the program. Alternately one can write the generator as a part of the program itself. The program transfers the data using *MPI_Send* and *MPI_Recv* for the last row of *hz* from each processor to the next and for the first row of *ey* from each processor to the previous (caring for boundary conditions). Modifying the loops from *ji* to *ij* also improved the performance by at least 25% (for 64 up to 300% for 1).

   Results:

   With Overheads:

Processors	1	2	4	8	16	32	64
Time (sec)	43.9	23.5	21.1	12.3	3.1	1.5	0.8
GFLOPS	0.4	0.7	0.8	1.3	4.3	10.4	20.4

   After removing overheads and optimizing:

Processors	1	2	4	8	16	32	64
Time (sec)	13.2	8.1	6.6	4.9	2.5	1.3	0.6
GFLOPS	1.2	2.0	2.4	3.2	6.4	12.7	25.7

   Min = -3.055Hz

   Max = 117.506Hz

   Mean = 0.866Hz

   RMS = 8.272Hz

Programs (removed unnecessary pieces of code)

1.

```
/* Find the first sqrt(n) primes */
  if(!id) {
    low_value = 2;
    high_value = (int)ceil((double) max_prime_init);
    size = high_value - low_value + 1;
    marked = (char *) malloc (size);

    for (i = 0; i < size; i++) marked[i] = 0;
    index = 0;
    prime = 2;
    do {
          first = prime * prime - low_value;

          for (i = first; i < size; i+= prime) marked[i] = 1;
          while (marked[++index]);
          prime = index + 2;
    } while (prime * prime <= max_prime_init);
    count = 0;
    for (i = 0; i < size; i++)
      if (!marked[i]) count++;
  }
/* Send count and allocate memory in each processor for storing sqrt(N) primes */
  MPI_Bcast(&count, 1, MPI_INT, 0, MPI_COMM_WORLD);
  primes = (int*) malloc(count * sizeof(int));
  if(!id) {
    j = 0;
    for (i = 0; i < size; i++)
      if (marked[i] == 0) {
        primes[j++] = i+2;
      }
  }
  nprimes = count;
/* Broadcast the sqrt(N) primes to all processors */
  MPI_Bcast(primes, count, MPI_INT, 0, MPI_COMM_WORLD);
  count = 0;
  max = 0;

  for ( j = 0; j < n; j+=chunk_size ) {
    low_value = j + 2 + id*(chunk_size-1)/p;
    if (j!=0) low_value--;
    high_value = j + 1 + (id+1)*(chunk_size-1)/p;
    size = high_value - low_value + 1;

    /* Allocate this process's share of the array. */
    marked = (char *) malloc (size);
```

```
    for (i = 0; i < size; i++) marked[i] = 0;
    if (!id) index = 0;

    for (i = 0; i < nprimes; i++) {
          prime = primes[i];
      if (prime * prime > low_value)
            first = prime * prime - low_value;
          else {
        if (!(low_value % prime)) first = 0;
            else first = prime - (low_value % prime);
          }
          for (k = first; k < size; k += prime) marked[k] = 1;
    }

/* Get count per processor and find last prime for max */
    for (i = 0; i < size; i++)
      if (!marked[i]) count++;
    for (i = size-1; i >= 0 ; i--)
      if (!marked[i]) { max = low_value + i; break; }
/* Reduce max ensuring that at least 1 prime has been found in chunk else use max from last chunk */
    if (p>1) MPI_Reduce (&max, &gmax, 1, MPI_INT, MPI_MAX,
      0, MPI_COMM_WORLD);
          else gmax = max;
    if ( gmax > global_max && !id )
      global_max = gmax;

  } //end for(j)
  /* Stop the timer */

  if (p > 1) MPI_Reduce (&count, &global_count, 1, MPI_INT, MPI_SUM,
    0, MPI_COMM_WORLD);
  else global_count = count;

  elapsed_time += MPI_Wtime();
```

2.

```
void compute_fdtd (int id, int p, double** ex, double** ey, double** hz, int N)
{
   int i, j, t;
   int size = BLOCK_SIZE(id,p,N);
   double hzp[N];
   double eyn[N];
   MPI_Status status;
   int err;

   for (t=0; t<Tmax; t++) {
/* Send last row of hz */
     if (id<p-1) err = MPI_Send (hz[size-1], N, MPI_TYPE, id+1, 0, MPI_COMM_WORLD);
     if (id) err = MPI_Recv (hzp, N, MPI_TYPE, id-1, 0, MPI_COMM_WORLD, &status);

/* compute the ey update for i = 0 */
     if (!id)
        for (j=0; j<N; j++)
           ey[0][j] = t;
     else
        for (j=0; j<N; j++) {
           ey[0][j] -= coeff1*(hz[0][j]-hzp[j]);
     for (i=1; i<size; i++)
        for (j=0; j<N; j++)
           ey[i][j] -= coeff1*(hz[i][j]-hz[i-1][j]);
     for (i=0; i<size; i++)
        for (j=1; j<N; j++)
            ex[i][j] -= coeff1*(hz[i][j]-hz[i][j-1]);

/* Send first row of ey */
     if (id) err = MPI_Send (ey[0], N, MPI_TYPE, id-1, 0, MPI_COMM_WORLD);
     if (id<p-1) err = MPI_Recv (eyn, N, MPI_TYPE, id+1, 0, MPI_COMM_WORLD, &status);

     for (i=0; i<size-1; i++)
        for (j=0; j<N-1; j++)
           hz[i][j] -= coeff2*(ex[i][j+1]-ex[i][j]+ey[i+1][j]-ey[i][j]);
/* Case for handling the last elements for hz */
     if (id<p-1)
        for (j=0; j<N-1; j++)
           hz[size-1][j] -= coeff2*(ex[size-1][j+1]-ex[size-1][j]+eyn[j]-ey[size-1][j]);

   }//for t

} //compute_ftdt

/* Check function to check for data validity */
void check(int size, double **hz, int N)
{
```

```
    double lmin, minval, lmax, maxval, lmean, mean, lrms, rms;
    int i, j;

    lmin = lmax = hz[0][0];
    lmean = lrms = 0.0;
    for (i=0; i<size; i++)
      for (j=0; j<N; j++) {
            if (hz[i][j] < lmin) lmin = hz[i][j];
            if (hz[i][j] > lmax) lmax = hz[i][j];
            lmean += hz[i][j]/(1.0*N*N);
            lrms  += hz[i][j]*hz[i][j]/(1.0*N*N);
          }
    if(p>1){
          MPI_Reduce(&lmin, &minval, 1, MPI_TYPE, MPI_MIN, 0, MPI_COMM_WORLD);
          MPI_Reduce(&lmax, &maxval, 1, MPI_TYPE, MPI_MAX, 0, MPI_COMM_WORLD);
          MPI_Reduce(&lmean, &mean, 1, MPI_TYPE, MPI_SUM, 0, MPI_COMM_WORLD);
          MPI_Reduce(&lrms, &rms, 1, MPI_TYPE, MPI_SUM, 0, MPI_COMM_WORLD);
    }
    else {
          minval = lmin;
          maxval = lmax;
          mean = lmean;
          rms = lrms;
    }
    rms = sqrt(rms);

    if(!id)
      printf("Minhz= %18.9f; Maxhz = %18.9f; Mean= %18.9f, RMS = %18.9f\n", minval, maxval, mean,
rms);
}
```