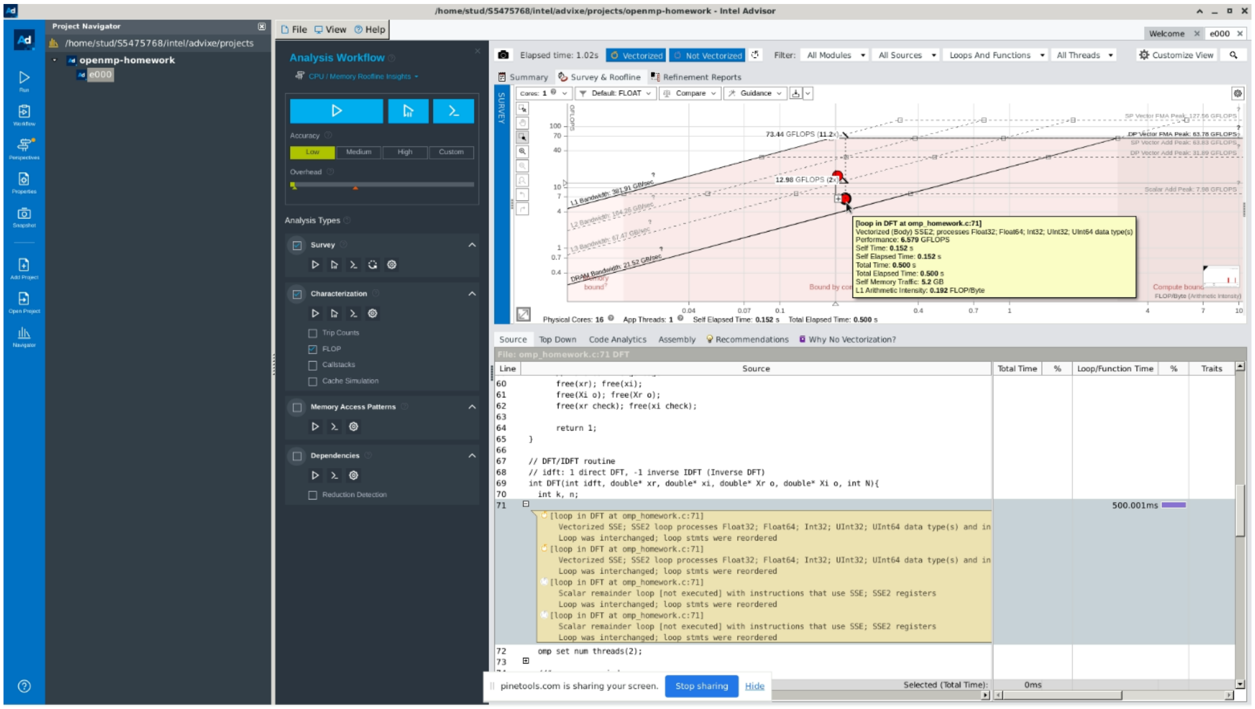
High-Performance Computing

Assignments

OpenMP

Hotspot identification



Elapsed time: 1.02s

[loop in DFT at omp_homework.c:71]
Vectorized (Body) SSE2; processes Float32; Float64; Int32; UInt32; UInt64 data type(s)
Performance: **6.579** GFLOPS
Self Time: **0.152** s
Self Elapsed Time: **0.152** s
Total Time: **0.500** s
Total Elapsed Time: **0.500** s
Self Memory Traffic: **5.2** GB
L1 Arithmetic Intensity: **0.192** FLOP/Byte

Physical Cores: 16 App Threads: 1 Self Elapsed Time: 0.152 s Total Elapsed Time: 0.500 s

File: omp_homework.c:71 DFT

```
        free(xr); free(xi);
        free(Xi o); free(Xr o);
        free(xr check); free(xi check);

        return 1;
    }

    // DFT/IDFT routine
    // idft: 1 direct DFT, -1 inverse IDFT (Inverse DFT)
    int DFT(int idft, double* xr, double* xi, double* Xr o, double* Xi o, int N){
      int k, n;
⊟
      [loop in DFT at omp_homework.c:71]
        Vectorized SSE; SSE2 loop processes Float32; Float64; Int32; UInt32; UInt64 data type(s) and in
        Loop was interchanged; loop stmts were reordered
      [loop in DFT at omp_homework.c:71]
        Vectorized SSE; SSE2 loop processes Float32; Float64; Int32; UInt32; UInt64 data type(s) and in
        Loop was interchanged; loop stmts were reordered
      [loop in DFT at omp_homework.c:71]
        Scalar remainder loop [not executed] with instructions that use SSE; SSE2 registers
        Loop was interchanged; loop stmts were reordered
      [loop in DFT at omp_homework.c:71]
        Scalar remainder loop [not executed] with instructions that use SSE; SSE2 registers
        Loop was interchanged; loop stmts were reordered
      omp set num threads(2);
⊞
```

500.001ms

pinetools.com is sharing your screen.

Selected (Total Time): 0ms

By using Intel Advisor (after compiling with *-g* flag) we can find that the hotspot is the loop by which we calculate the values of the Discrete Fourier Transform, also by manually reviewing the code we can spot the same.

```
for (k=0 ; k<N ; k++)
{
   for (n=0 ; n<N ; n++)  {
     // Real part of X[k]
    Xr_o[k] += xr[n] * cos(n * k * PI2 / N) + idft*xi[n]*sin(n *   k * 
PI2 / N);
     // Imaginary part of X[k]
    Xi_o[k] += -idft*xr[n] * sin(n * k * PI2 / N) + xi[n] * cos(n * k 
* PI2 / N);
  }
}
```

Vectorization

To understand which loops have been vectorized and which have not, we compiled the code with optimization level -O2 and report level 5.

```
icc  -O2 -qopenmp -qopt-report=5 -qopt-report-phase=vec omp_homework.c
-o exc.o
```

We will focus on some of the vectorization issues highlighted by the report:

```
Begin optimization report for: DFT(int, double *, double *, double *,
double *, int)

    Report from: Vector optimizations [vec]


LOOP BEGIN at omp_homework.c(78,3)
   remark #15541: outer loop was not auto-vectorized: consider using
SIMD directive

   LOOP BEGIN at omp_homework.c(80,7)
      remark #15344: loop was not vectorized: vector dependence
prevents vectorization
      remark #15346: vector dependence: assumed OUTPUT dependence
between Xr_o[k] (82:11) and Xi_o[k] (84:11)
      remark #15346: vector dependence: assumed OUTPUT dependence
between Xi_o[k] (84:11) and Xr_o[k] (82:11)
   LOOP END
LOOP END
```

The outer loop has not been vectorized and the vectorization report suggests to use:

```
#pragma omp simd
```

The inner loop could not be vectorized due to dependencies between instructions.

```
LOOP BEGIN at omp_homework.c(105,3)
   remark #15527: loop was not vectorized: function call to rand(void)
cannot be vectorized   [ omp_homework.c(106,5) ]
LOOP END
```

Loop with function call inside, could not be vectorized.

```
Begin optimization report for: checkResults(double *, double *, double
*, double *, double *, double *, int)

    Report from: Vector optimizations [vec]


LOOP BEGIN at omp_homework.c(131,3)
   remark #15382: vectorization support: call to function printf(const
char *__restrict__, ...) cannot be vectorized   [
omp_homework.c(133,7) ]
```

```
   remark #15382: vectorization support: call to function printf(const
char *__restrict__, ...) cannot be vectorized   [
omp_homework.c(135,7) ]
   remark #15344: loop was not vectorized: vector dependence prevents
vectorization
LOOP END
```

The loop was not vectorized because of the call of *printf* Function inside the loop, which prevents vectorization.

Parallelization

```
int DFT(int idft, double* xr, double* xi, double* Xr_o, double* Xi_o,
int N){

  int k, n;

  omp_set_num_threads(24);
  #pragma omp parallel for  private(k, n) shared(xr, xi, Xr_o, Xi_o)
schedule(static)
  for (k=0 ; k<N ; k++)
  {
      for (n=0 ; n<N ; n++)  {
        // Real part of X[k]
          Xr_o[k] += xr[n] * cos(n * k * PI2 / N) + idft*xi[n]*sin(n *
k * PI2 / N);
          // Imaginary part of X[k]
          Xi_o[k] += -idft*xr[n] * sin(n * k * PI2 / N) + xi[n] *
cos(n * k * PI2 / N);

      }
  }

  // normalize if you are doing IDFT
  if (idft==-1){
    for (n=0 ; n<N ; n++){
      Xr_o[n] /=N;
      Xi_o[n] /=N;
    }
  }
  return 1;
}
```

We should pay attention to the *shared* and *private* variables between threads, where arrays should be shared between threads while indexes should be private. We used static scheduling due to the workload-balanced nature of the problem between threads.

Experiments

Configuration	Time Elapsed (seconds)
Serial code with -O2	1.023963
Serial code with #pragma omp simd	1.048658
Threads = 2	0.520456
Threads = 4	0.267030
Threads = 8	0.231641
Threads = 16	0.163901
Threads = 24	0.113265

We can observe that the speed up is *sub-linear* and thread efficiency is low after exceeding 4 threads.

CUDA

In CUDA we need to develop a kernel that will be executed by each thread:

```
__global__ void step_kernel_mod(int ni, int nj, float fact, float*
temp_in, float* temp_out)
{
  int i00, im10, ip10, i0m1, i0p1;
  float d2tdx2, d2tdy2;

  //evaluate i and j
  // loop over all points in domain (except boundary)
  int i = blockIdx.x * blockDim.x + threadIdx.x; //column
  int j = blockIdx.y * blockDim.y + threadIdx.y; //row
  if (i < ni-1 && i > 0 && j > 0 && j< nj-1){
      // find indices into linear memory
      // for central point and neighbours
      //this way of finding the exact locations is not perfict since
wee need to calculate the derivatives
      /*i00 = j * gridDim.x * blockDim.x + i;
      im10 = j * gridDim.x * blockDim.x + (i-1);
      ip10 = j * gridDim.x * blockDim.x + (i+1);
      i0m1 = (j-1) * gridDim.x * blockDim.x + i;
      i0p1 = (j+1) * gridDim.x * blockDim.x + i;*/
      i00 = I2D(ni, i, j);
      im10 = I2D(ni, i-1, j);
      ip10 = I2D(ni, i+1, j);
      i0m1 = I2D(ni, i, j-1);
      i0p1 = I2D(ni, i, j+1);

      // evaluate derivatives
      d2tdx2 = temp_in[im10]-2*temp_in[i00]+temp_in[ip10];
      d2tdy2 = temp_in[i0m1]-2*temp_in[i00]+temp_in[i0p1];

      // update temperatures
      temp_out[i00] = temp_in[i00]+fact*(d2tdx2 + d2tdy2);
    }
}
```

We need to pay attention inside the kernel to process the right data, so we check before processing, like excluding boundaries. Since we are doing second derivates along the two axes we need to query the right data points, wrong querying will lead to wrong results.

We developed a 2D grid with 2D Blocks inside, this structure will help in indexing and querying the right data points inside each thread since they are mounted in a big 2D grid.

We need to allocate development arrays in Cuda Memory and copy initialization values to them, after calling the kernel we need to copy back the results from the development arrays to arrays in the host. Copy will force synchronization.

```
cudaMalloc((void**)&temp1_dev, size);
  cudaMalloc((void**)&temp2_dev, size);
```

```

  dim3 blocksPerGrid((ni + nthreads -1)/nthreads,(nj + nthreads -
1)/nthreads);
  dim3 threadsPerBlock(nthreads, nthreads);
  cout<<"GRID DIM_1: "<<blocksPerGrid.x<<" GRID DIM_2:
"<<blocksPerGrid.y<<endl;
  cout<<"BLOCK DIM 1: "<<threadsPerBlock.x<<" BLOCK DIM_2:
"<<threadsPerBlock.y<<endl;

  //copy the intialization of tempreture values to device
    cudaMemcpy(temp1_dev, temp1, size, cudaMemcpyHostToDevice);
    cudaMemcpy(temp2_dev, temp2, size, cudaMemcpyHostToDevice);

  for (istep=0; istep < nstep; istep++) {

    step_kernel_mod<<<blocksPerGrid, threadsPerBlock>>>(ni, nj, tfac,
temp1_dev, temp2_dev);
    cudaDeviceSynchronize();
    // swap the temperature pointers
    temp_tmp = temp1_dev;
    temp1_dev = temp2_dev;
    temp2_dev= temp_tmp;
  }
  cudaMemcpy(temp1, temp1_dev, size, cudaMemcpyDeviceToHost);
  cudaMemcpy(temp2, temp2_dev, size, cudaMemcpyDeviceToHost);
```

The number of threads per block is a parameter we use to configure the grid dimensions.

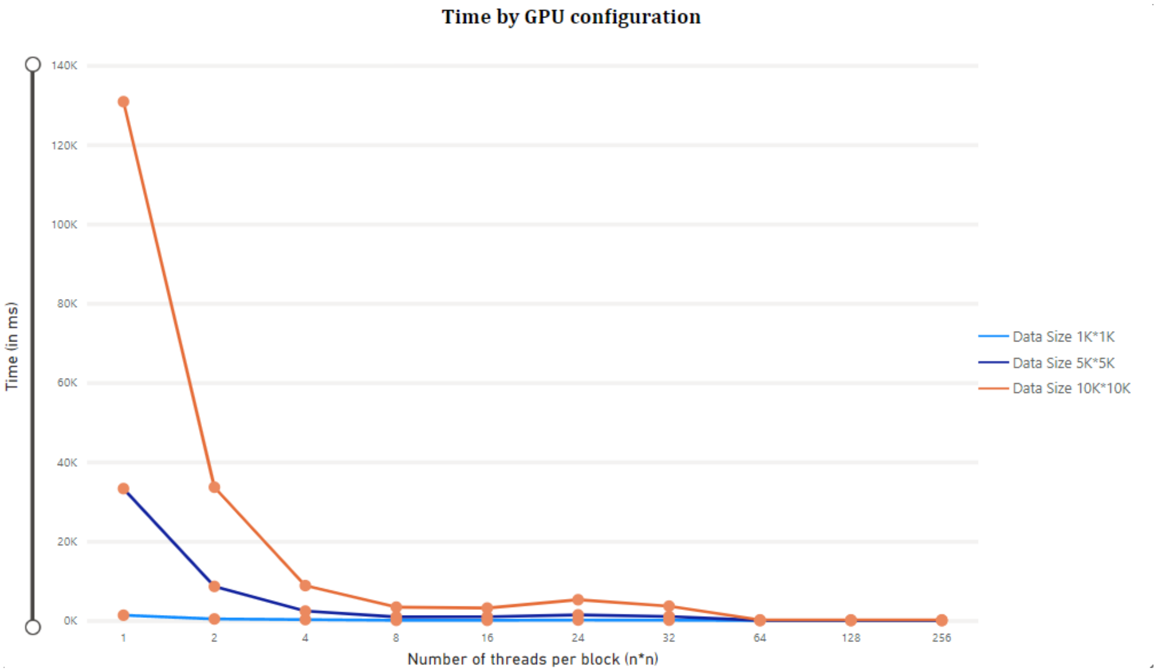
Experiments

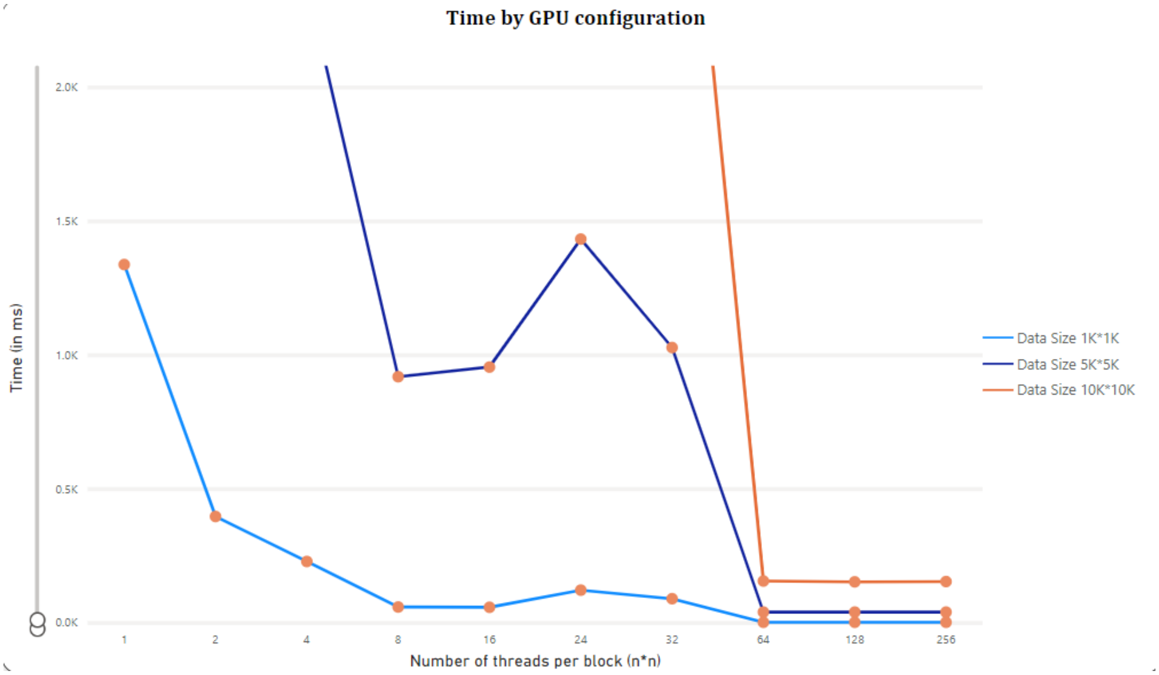
We tried different Grid/block dimension configurations with different problem sizes.

Data Size	Grid DIM	Block DIM	Time in MS
1K*1K	4*4	256*256	1
1K*1K	8*8	128*128	1
1K*1K	16*16	64*64	1
1K*1K	32*32	32*32	89
1K*1K	42*42	24*24	121
1K*1K	63*63	16*16	57
1K*1K	125*125	8*8	58
1K*1K	250*250	4*4	228
1K*1K	500*500	2*2	396
1K*1K	1000*1000	1*1	1337
5K*5K	20*20	256*256	39
5K*5K	40*40	128*128	39
5K*5K	79*79	64*64	39
5K*5K	157*157	32*32	1027
5K*5K	209*209	24*24	1432
5K*5K	313*313	16*16	954
5K*5K	625*625	8*8	918
5K*5K	1250*1250	4*4	2385
5K*5K	2500*2500	2*2	8601
5K*5K	5k*5k	1*1	33272
10K*10K	40*40	256*256	153
10K*10K	79*79	128*128	152
10K*10K	157*157	64*64	155
10K*10K	209*209	32*32	3633
10K*10K	313*313	24*24	5248
10K*10K	625*625	16*16	3144
10K*10K	1250*1250	8*8	3375
10K*10K	2500*2500	4*4	8807
10K*10K	5k*5k	2*2	33621
10K*10K	10k*10K	1*1	130857

We tried 30K * 30K data size, but we got the following error (due to the large amount of data):

```
Segmentation fault
```





We can notice that the best threads per block configuration are (64, 64), (128, 128) and (256, 256) where the time elapsed is the same among all of them for each problem size. It is worth mentioning that the wrap size in the utilized GPU is /32/.

MPI

In MPI the workload will be distributed among different processes, work duty for each process will be decided based on the number of processes, while the process rank will define the duty boundaries.

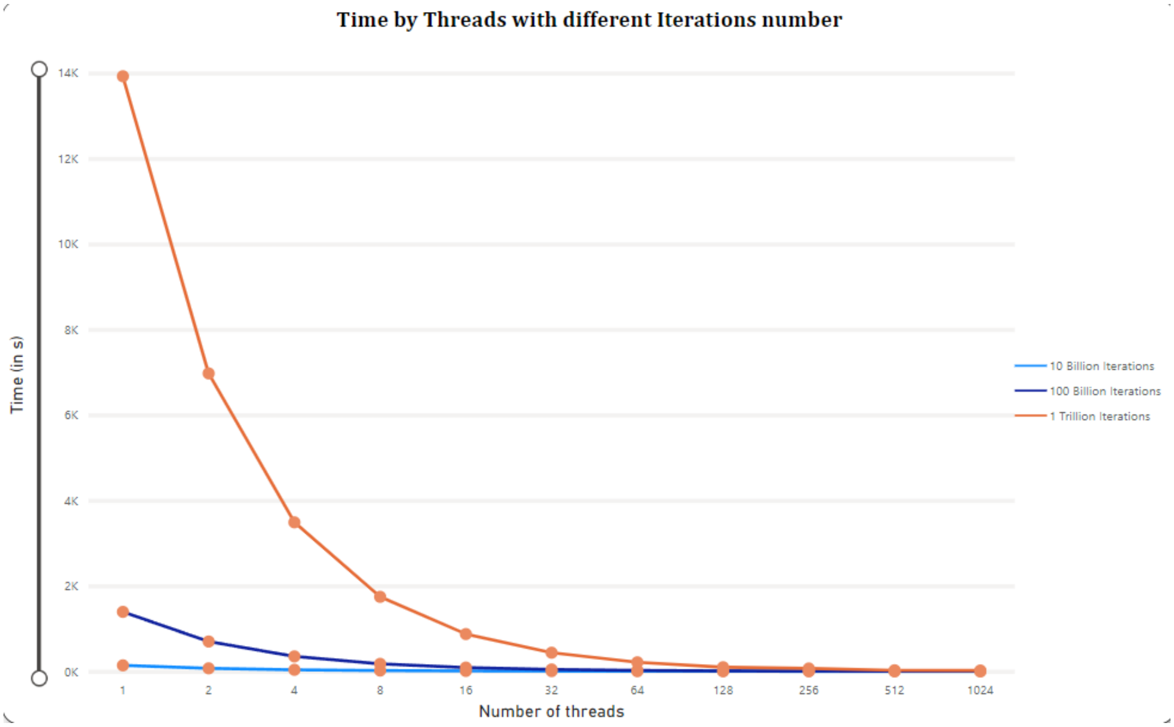
```
 int myid, numprocs;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);



    long dutySize =  intervals / numprocs;
    long int startPos = myid * dutySize + 1;
    long int endPos = startPos + dutySize;

    if(myid == 0){
        printf("Number of intervals: %ld\n", intervals);
        std::cout<<"Number of Processes: "<<numprocs<<std::endl;
        std::cout<<"Duty Size for each Process is: 
"<<dutySize<<std::endl;
    }

    long int i;
    double x, dx, f, sum, globalSum, pi;
    double time2;

    time_t time1 = clock();

    sum = 0.0;
    dx = 1.0 / (double) intervals;

    for (i = startPos; i < endPos; i++) {
        x = dx * ((double) (i - 0.5));
        f = 4.0 / (1.0 + x*x);
        sum = sum + f;
    }

    MPI_Reduce(&sum, &globalSum, 1, MPI_DOUBLE, MPI_SUM, 0, 
MPI_COMM_WORLD);

    if(myid == 0){
        pi = dx*globalSum;
        time2 = (clock() - time1) / (double) CLOCKS_PER_SEC;
        printf("Computed PI %.24f\n", pi);
        printf("The true PI %.24f\n\n", PI25DT);
```

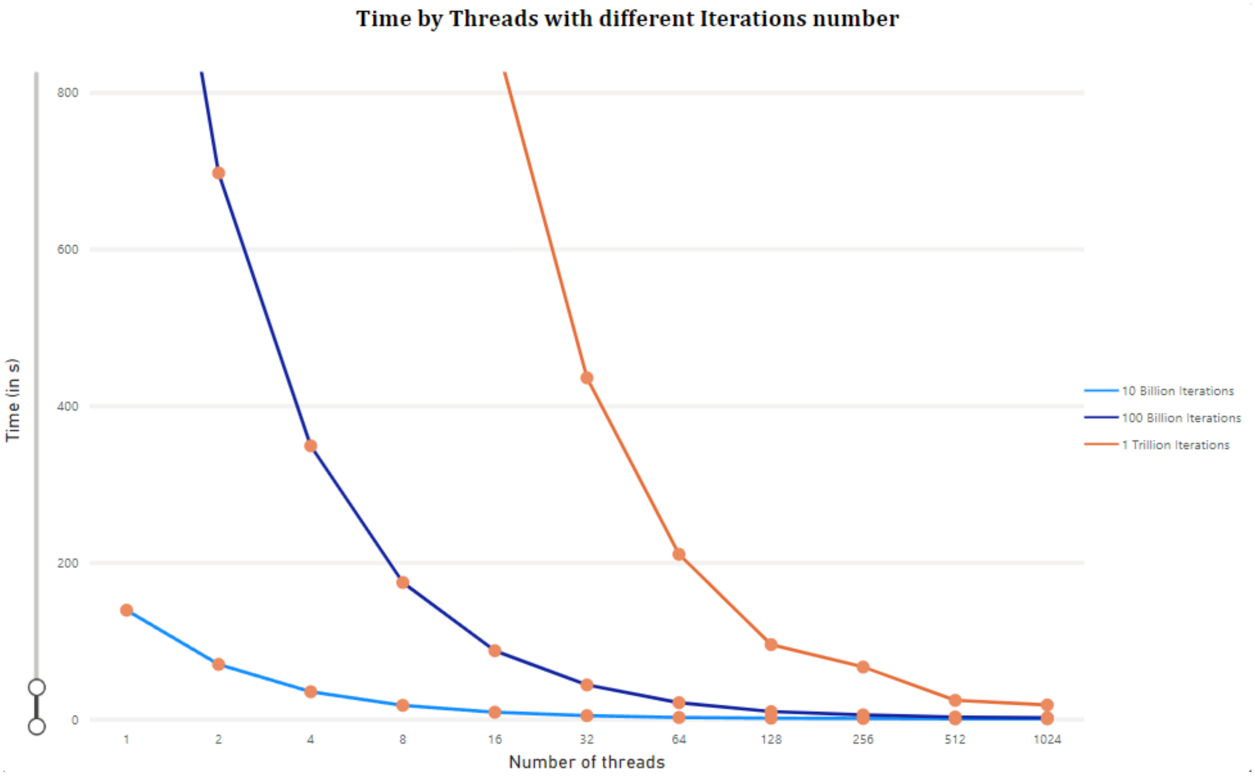
Each process will calculate a partial sum to be reduced later by the root process.

Experiments

We tried to use different Iterations with different processes count.

Number of Processes	10 Billion Iterations Time Elapsed (SEC)	100 Billion Iterations Time Elapsed (SEC)	1 Trillion Iterations Time Elapsed (SEC)
1024	0.26	1.79	17.93
512	0.25	2.65	23.88
256	0.71	5.34	66.41
128	1.09	9.46	94.97
64	2.02	20.88	210.34
32	4.34	43.55	435.49
16	8.71	87.21	871.52
8	17.44	174.32	1742.87
4	34.87	348.68	3485.74
2	69.74	696.97	6970.96
1	138.93	1390.91	13924.2





We can notice that the speed-up was almost linear till reaching /512/ processes, after that increasing processes to /1024/ does not affect the execution time anymore. Message passing overhead affects the execution time, and we assume, it is the reason behind stopping the enhancement.