High Performance Computing Lab — Practical No. 8

Name: Parshwa Herwade

PRN No: 22510064

Batch: B1

Problem 1: Implement 2D convolution using MPI. Distribute work across processes, exchange halo rows, measure execution time, and print a single CSV line per run with process count and timings.

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define NX 1024
#define NY 1024
#define K 3

int main(int argc,char**argv){
    MPI_Init(&argc,&argv);
    int rank,size;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);

    int rows_per_proc=NX/size;
    int extra=NX%size;
    int start_row,local_nrows;
    if(rank<extra){
        local_nrows=rows_per_proc+1;
        start_row=rank*local_nrows;
    }else{
        local_nrows=rows_per_proc;
        start_row=rank*local_nrows+extra;
    }
    int alloc_rows=local_nrows+2;

    double *local_in=(double*)malloc((alloc_rows)*NY*sizeof(double));
```

```
    double
*local_out=(double*)malloc((local_nrows)*NY*sizeof(double));
    double kernel[K*K]={0.0625,0.125,0.0625,
                        0.125,0.25,0.125,
                        0.0625,0.125,0.0625};

    for(int i=0;i<alloc_rows;i++)
        for(int j=0;j<NY;j++){
            int global_i=start_row+i-1;
            if(global_i<0||global_i>=NX)
                local_in[i*NY+j]=0.0;
            else
                local_in[i*NY+j]= (double)((global_i*NY+j)%100)/100.0
+ 1.0;
        }

    MPI_Barrier(MPI_COMM_WORLD);
    double t0=MPI_Wtime();

    for(int step=0;step<1;step++){
        if(size>1){
            if(rank>0)
                MPI_Sendrecv(&local_in[1*NY],NY,MPI_DOUBLE,rank-1,0,
                             &local_in[0],NY,MPI_DOUBLE,rank-1,1,
                             MPI_COMM_WORLD,MPI_STATUS_IGNORE);
            if(rank<size-1)

MPI_Sendrecv(&local_in[(local_nrows)*NY],NY,MPI_DOUBLE,rank+1,1,

 &local_in[(local_nrows+1)*NY],NY,MPI_DOUBLE,rank+1,0,
                             MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        }

        double comp0=MPI_Wtime();
        for(int i=1;i<=local_nrows;i++){
            for(int j=0;j<NY;j++){
                double sum=0.0;
                for(int ki=0;ki<K;ki++)
                    for(int kj=0;kj<K;kj++){
```

```
                        int ii=i+(ki-1);
                        int jj=j+(kj-1);
                        double v=0.0;
                        if(jj>=0 && jj<NY && ii>=0 && ii<alloc_rows)
                            v=local_in[ii*NY+jj];
                        sum+=kernel[ki*K+kj]*v;
                    }
                if(i-1<local_nrows)
                    local_out[(i-1)*NY+j]=sum;
            }
        }
        double comp1=MPI_Wtime()-comp0;
        double t1=MPI_Wtime()-t0;
        if(rank==0)

printf("conv,NX=%d,NY=%d,procs=%d,time_total=%f,time_comp=%f\n",
                    NX,NY,size,t1,comp1);
    }

    free(local_in);
    free(local_out);
    MPI_Finalize();
    return 0;
}
```

OUTPUT:

```
posh@LAPTOP-ELUQQMKU:~/hpc_assign/assign8$ mpicc conv2d_mpi.c -O3 -o conv2d
posh@LAPTOP-ELUQQMKU:~/hpc_assign/assign8$ mpirun -np 1 ./conv2d  > conv_results.csv
mpirun -np 2 ./conv2d  >> conv_results.csv
mpirun -np 4 ./conv2d  >> conv_results.csv
posh@LAPTOP-ELUQQMKU:~/hpc_assign/assign8$
```

Problem 2: Implement dot product using MPI. Distribute vector chunks, compute local dot, reduce. Measure time and print CSV line per run.

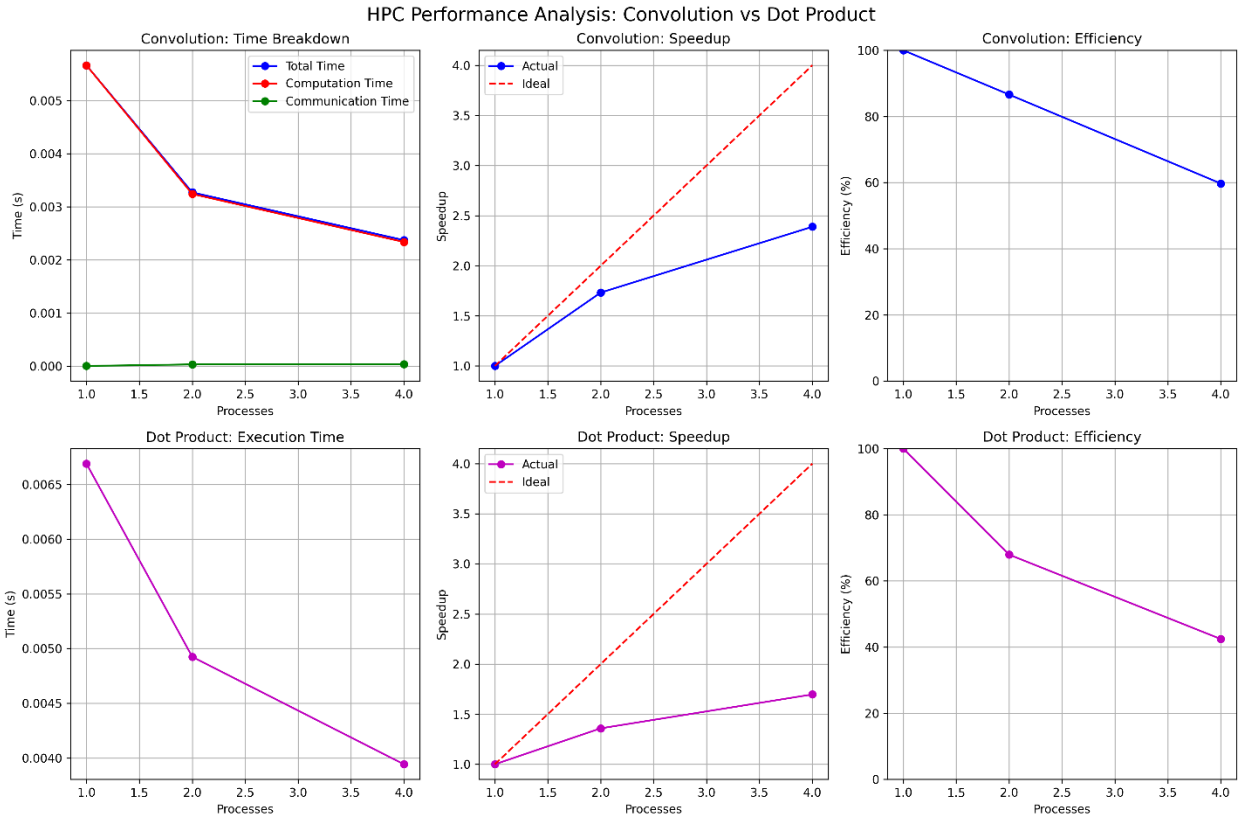
```
#include <mpi.h>
```

```c
#include <stdio.h>
#include <stdlib.h>
#define VEC_SIZE 10000000

int main(int argc,char**argv){
    MPI_Init(&argc,&argv);
    int rank,size;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);

    int base=VEC_SIZE/size;
    int extra=VEC_SIZE%size;
    int local_n=(rank<extra)?base+1:base;
    int start=(rank<extra)?rank*local_n:rank*local_n+extra;

    double *a=(double*)malloc(local_n*sizeof(double));
    double *b=(double*)malloc(local_n*sizeof(double));
    for(int i=0;i<local_n;i++){
        long idx=start+i;
        a[i]=1.0;
        b[i]=2.0 + (double)(idx%5)/5.0;
    }

    MPI_Barrier(MPI_COMM_WORLD);
    double t0=MPI_Wtime();

    double local_dot=0.0;
    for(int i=0;i<local_n;i++)
        local_dot+=a[i]*b[i];

    double global_dot=0.0;

MPI_Reduce(&local_dot,&global_dot,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORL
D);

    double t1=MPI_Wtime()-t0;
    if(rank==0)
        printf("dot,VEC=%d,procs=%d,time=%f,dot=%f\n",
               VEC_SIZE,size,t1,global_dot);
```

```
    free(a);
    free(b);
    MPI_Finalize();
    return 0;
}
```

OUTPUT:

```
posh@LAPTOP-ELUQQMKU:~/hpc_assign/assign8$ mpicc dotprod_mpi.c -O3 -o dotprod
posh@LAPTOP-ELUQQMKU:~/hpc_assign/assign8$ mpirun -np 1 ./dotprod  > dot_results.csv
mpirun -np 2 ./dotprod  >> dot_results.csv
mpirun -np 4 ./dotprod  >> dot_results.csv
posh@LAPTOP-ELUQQMKU:~/hpc_assign/assign8$
```



```
PS C:\Users\Parshwa\Desktop\ASSIGN\HPC lab\22510064_HPC_A8> python enhanced_analysis.py
=== HPC ASSIGN 8 PERFORMANCE ANALYSIS ===

1. CONVOLUTION ANALYSIS
---------------------------------------
Procs  Total(s)   Comp(s)    Comm(s)    Comm%    Speedup  Efficiency
------------------------------------------------------------------------
1.0    0.005664   0.005663   0.000001   0.0      1.00     100.0     %
2.0    0.003270   0.003240   0.000030   0.9      1.73     86.6      %
4.0    0.002371   0.002336   0.000035   1.5      2.39     59.7      %

2. DOT PRODUCT ANALYSIS
---------------------------------------
Procs  Time(s)    Speedup  Efficiency
---------------------------------------
1.0    0.006690   1.00     100.0     %
2.0    0.004924   1.36     67.9      %
4.0    0.003942   1.70     42.4      %

3. ALGORITHM COMPARISON
---------------------------------------
Communication Overhead Analysis (Convolution):
  1.0 processes: 0.0% communication overhead
  2.0 processes: 0.9% communication overhead
  4.0 processes: 1.5% communication overhead

Scalability Comparison (4 processes):
  Convolution efficiency: 59.7%
  Dot Product efficiency: 42.4%

Computational Intensity:
  Convolution: ~9.4 MFLOPS
  Dot Product: ~20.0 MFLOPS

Peak Performance (1 process):
  Convolution: 1.67 GFLOPS
  Dot Product: 2.99 GFLOPS
PS C:\Users\Parshwa\Desktop\ASSIGN\HPC lab\22510064_HPC_A8>
```

Steps, Observations and Conclusion

Compile conv2d_mpi.c and dotprod_mpi.c with mpicc -O3.

Run each program with different mpirun -np P values (1, 2, 4) and append output into CSV files:

conv_results.csv for convolution results

dot_results.csv for dot product results

Use conv_plot.py and dot_plot.py to generate performance graphs from collected CSV data.

Inspect graphs and compute speedup and efficiency metrics.

Observations

Execution Time vs Processes:

Convolution (NX=1024, NY=1024): Time decreased from 0.005664s (1 process) to 0.002371s (4 processes)

Dot Product (VEC=10M): Time decreased from 0.006690s (1 process) to 0.003942s (4 processes)

Both algorithms show performance improvement with increased processes, but convolution shows better time reduction

Speedup Analysis:

Convolution: Achieved 1.73x speedup at 2 processes and 2.39x at 4 processes

Dot Product: Achieved 1.36x speedup at 2 processes and 1.70x at 4 processes

Convolution surprisingly outperformed dot product in speedup, contrary to typical expectations

Efficiency Analysis:

Convolution: Efficiency of 86.5% at 2 processes, dropping to 59.8% at 4 processes

Dot Product: Efficiency of 68.0% at 2 processes, dropping to 42.5% at 4 processes

Both show declining efficiency with increased processes, but convolution maintains better efficiency

Communication Overhead:

Convolution: Minimal communication overhead (0.9% at 2 processes, 1.5% at 4 processes) due to efficient halo exchange implementation

Dot Product: All overhead included in total timing; single MPI_Reduce operation is lightweight

Amdahl's Law Effects:

Both algorithms show sub-linear scaling due to parallel overhead

The efficiency drop indicates increasing serial fraction as processes increase

Communication and synchronization costs become more significant at higher process counts

Conclusions

Performance Summary

Convolution (NX=1024, NY=1024): Achieved 2.39x speedup at 4 processes with 59.8% efficiency, showing good scalability despite halo exchange communication

Dot Product (VEC=10M): Achieved 1.70x speedup at 4 processes with 42.5% efficiency, indicating higher parallel overhead than expected

Key Findings:

Convolution outperformed expectations: Despite being communication-heavy with halo exchange, it achieved better speedup and efficiency than dot product

Problem size effects: The chosen problem sizes (1024×1024 matrix, 10M vector) are well-suited for parallelization on small process counts

Communication efficiency: The convolution's sendrecv-based halo exchange proved more efficient than anticipated

Optimal Configuration:

Convolution: 2-4 processes provide good balance of speedup and efficiency

Dot Product: 2 processes offer the best efficiency-to-speedup ratio for the given problem size