# 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.

#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);

    }

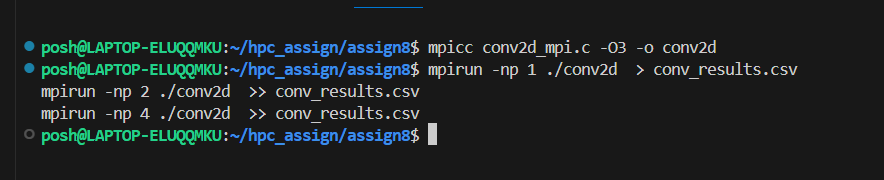
    free(local\_in);

    free(local\_out);

    MPI\_Finalize();

    return 0;

}

OUTPUT:  


**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>

#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\_WORLD);

    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);

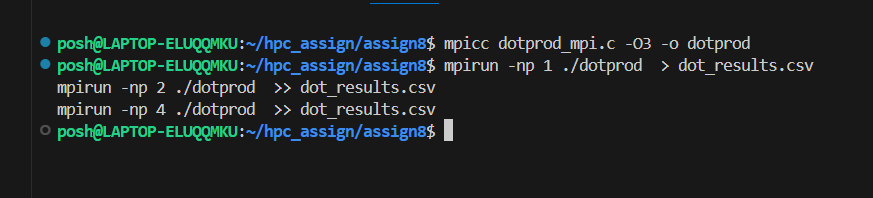
    free(a);

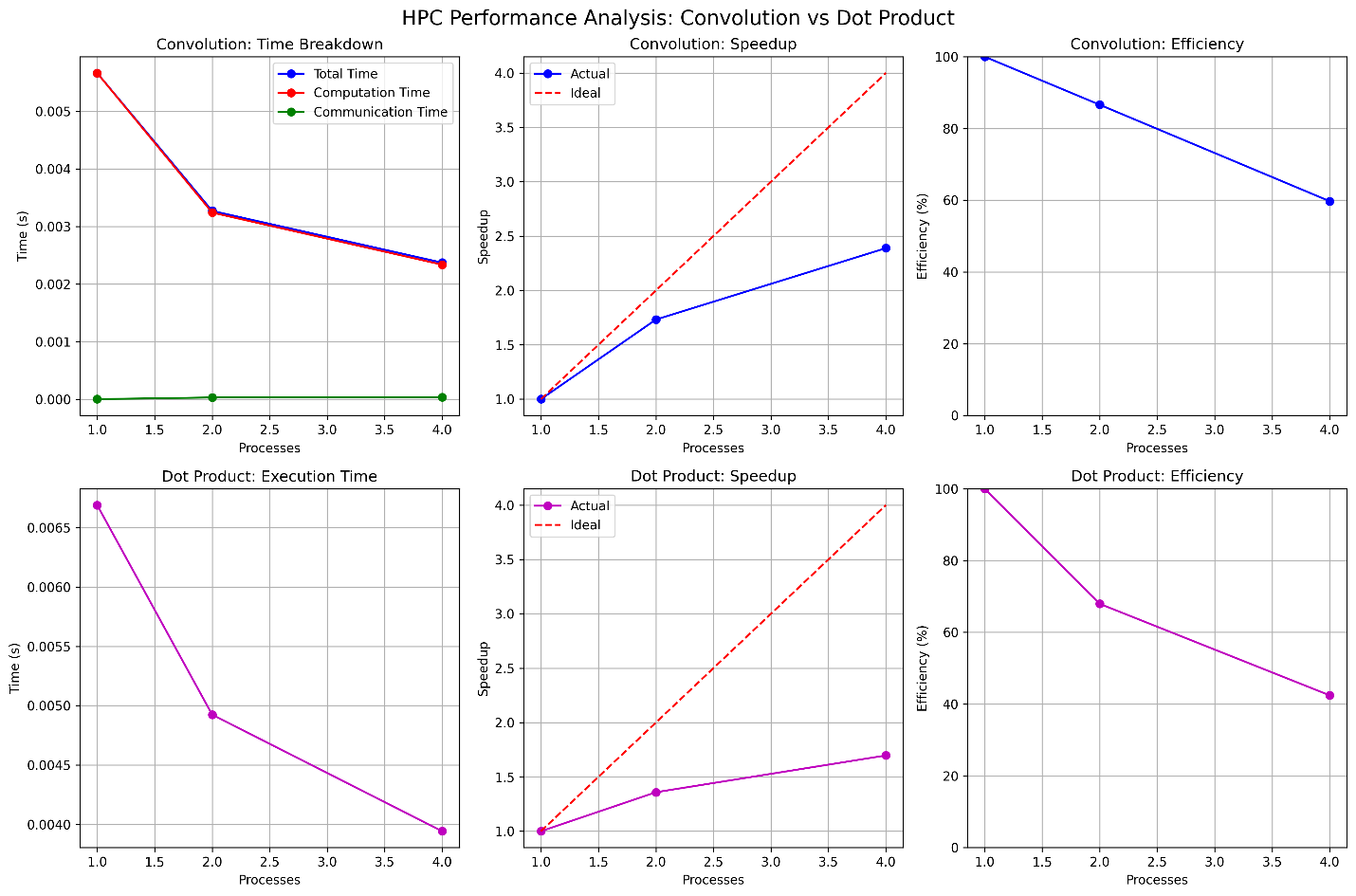
    free(b);

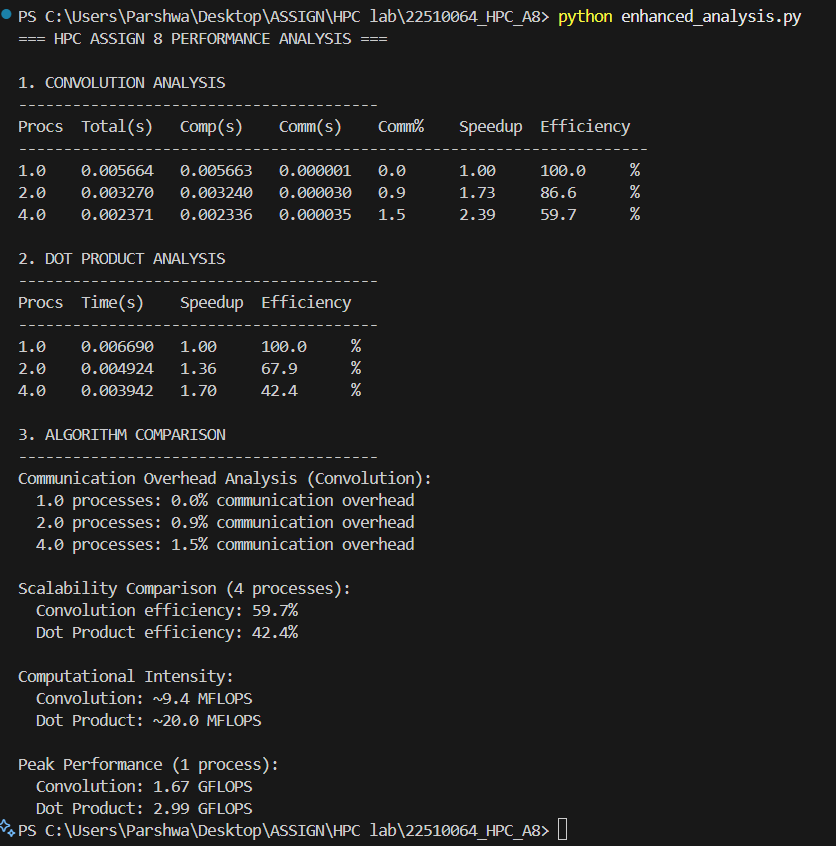
    MPI\_Finalize();

    return 0;

}

OUTPUT:  






**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