Date: 23/11/2022

High Performance Computing Lab

Assignment - 8

PRN: 2019BTECS00029
Name: HarshalKodgire

MPI Programming

1. Study and implement 2D Convolution using MPI. Use different number of processes and analyze the performance.

   Code:

```
#include<assert.h>
#include<mpi.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/time.h>

#defineDEFAULT_ITERATIONS1

intconv_column(int*,int,int,int,int*,int);
intconv(int*,int,int,int,int*,int);
int*check(int*,int,int,int*,int);

intconv_column(int*sub_grid,inti,intnrows,int
DIM,int*kernel,intkernel_dim) {
intcounter=0;
```

```
intnum_pads= (kernel_dim-1) /2;

for (intj=1; j< (num_pads+1); j++) {
counter=counter+sub_grid[i+j*DIM]
*kernel[(((kernel_dim-1) * (kernel_dim+1)) /2)
+j*kernel_dim];
counter=counter+sub_grid[i-j*DIM]
*kernel[(((kernel_dim-1) * (kernel_dim+1)) /2) -
j*kernel_dim];
    }
counter=counter+sub_grid[i] *kernel[(((kernel_dim-1)
* (kernel_dim+1)) /2)];

returncounter;
}

intconv(int*sub_grid,inti,intnrows,int
DIM,int*kernel,intkernel_dim) {
intcounter=0;
intnum_pads= (kernel_dim-1) /2;
// convolve middle column
counter=counter+conv_column(sub_grid, i, nrows, DIM,
kernel, kernel_dim);

// convolve left and right columns
for (intj=1; j< (num_pads+1); j++) {
// get last element of current row
intend= (((i/DIM) +1) *DIM) -1;
if (i+j-end<=0) { // if column is valid
counter=counter+conv_column(sub_grid, i+j, nrows,
DIM, kernel, kernel_dim);
        }
```

```
// get first element of current row
intfirst= (i/DIM) *DIM;
if (i-j-first>=0) {
counter=counter+conv_column(sub_grid, i-j, nrows,
DIM, kernel, kernel_dim);
        }
    }

returncounter;
}

int*check(int*sub_grid,intnrows,int
DIM,int*kernel,intkernel_dim) {
intval;
intnum_pads= (kernel_dim-1) /2;
int*new_grid=calloc(DIM*nrows, sizeof(int));
for (inti= (num_pads*DIM); i< (DIM*
(num_pads+nrows)); i++) {
val=conv(sub_grid, i, nrows, DIM, kernel,
kernel_dim);
new_grid[i- (num_pads*DIM)] =val;
    }
returnnew_grid;
}

intmain(intargc,char**argv) {
// MPI Standard variable
intnum_procs;
intID, j;
intiters=0;
intnum_iterations;
intDIM;
```

```
intGRID_WIDTH;
intKERNEL_DIM;
intKERNEL_SIZE;

num_iterations=DEFAULT_ITERATIONS;
if (argc>=3) {
DIM=atoi(argv[1]);
GRID_WIDTH=DIM*DIM;
KERNEL_DIM=atoi(argv[2]);
KERNEL_SIZE=KERNEL_DIM*KERNEL_DIM;
if (argc==4) {
num_iterations=atoi(argv[3]);
        }
    } else {
printf("Invalid command line arguments");
MPI_Finalize();
exit(-1);
    }
intmain_grid[GRID_WIDTH];
memset(main_grid, 0, GRID_WIDTH*sizeof(int));
for (inti=0; i<GRID_WIDTH; i++) {
main_grid[i] =1;
    }

intnum_pads= (KERNEL_DIM-1) /2;

intkernel[KERNEL_SIZE];
memset(kernel, 0, KERNEL_SIZE*sizeof(int));
for (inti=0; i<KERNEL_SIZE; i++) {
kernel[i] =1;
    }
// Messaging variables
```

```
MPI_Statusstatus;

// MPI Setup
MPI_Init(NULL, NULL);
//    if ( MPI_Init( &argc, &argv ) != MPI_SUCCESS )
//    {
//      printf( "MPI_Init error\n" );
//    }

MPI_Comm_size(MPI_COMM_WORLD, &num_procs); // Set the
num_procs
MPI_Comm_rank(MPI_COMM_WORLD, &ID);

doublestart_time=MPI_Wtime();
assert(DIM%num_procs==0);

intupper[DIM*num_pads];
intlower[DIM*num_pads];

int*pad_row_upper;
int*pad_row_lower;

intstart= (DIM/num_procs) *ID;
intend= (DIM/num_procs) -1+start;
intnrows=end+1-start;
intnext= (ID+1) %num_procs;
intprev=ID!=0?ID-1:num_procs-1;

for (iters=0; iters<num_iterations; iters++) {

memcpy(lower, &main_grid[DIM* (end-num_pads+1)],
sizeof(int) *DIM*num_pads);
```

```
pad_row_lower=malloc(sizeof(int) *DIM*num_pads);

memcpy(upper, &main_grid[DIM*start], sizeof(int)
*DIM*num_pads);
pad_row_upper=malloc(sizeof(int) *DIM*num_pads);

if (num_procs>1) {
if (ID%2==1) {
MPI_Recv(pad_row_lower, DIM*num_pads, MPI_INT, next,
1, MPI_COMM_WORLD, &status);
MPI_Recv(pad_row_upper, DIM*num_pads, MPI_INT, prev,
1, MPI_COMM_WORLD, &status);
            } else {
MPI_Send(upper, DIM*num_pads, MPI_INT, prev, 1,
MPI_COMM_WORLD);
MPI_Send(lower, DIM*num_pads, MPI_INT, next, 1,
MPI_COMM_WORLD);
            }
if (ID%2==1) {
MPI_Send(upper, DIM*num_pads, MPI_INT, prev, 0,
MPI_COMM_WORLD);
MPI_Send(lower, DIM*num_pads, MPI_INT, next, 0,
MPI_COMM_WORLD);
            } else {
MPI_Recv(pad_row_lower, DIM*num_pads, MPI_INT, next,
0, MPI_COMM_WORLD, &status);
MPI_Recv(pad_row_upper, DIM*num_pads, MPI_INT, prev,
0, MPI_COMM_WORLD, &status);
            }
        } else {
pad_row_lower=upper;
pad_row_upper=lower;
```

```
        }

intsub_grid[DIM* (nrows+ (2*num_pads))];
if (ID==0) {
memset(pad_row_upper, 0, DIM*sizeof(int) *num_pads);
        }
if (ID== (num_procs-1)) {
memset(pad_row_lower, 0, DIM*sizeof(int) *num_pads);
        }
memcpy(sub_grid, pad_row_upper, sizeof(int)
*DIM*num_pads);
memcpy(&sub_grid[DIM*num_pads],
&main_grid[DIM*start], sizeof(int) *DIM*nrows);
memcpy(&sub_grid[DIM* (nrows+num_pads)],
pad_row_lower, sizeof(int) *DIM*num_pads);
int*changed_subgrid=check(sub_grid, nrows, DIM,
kernel, KERNEL_DIM);

if (ID!=0) {
MPI_Send(changed_subgrid, nrows*DIM, MPI_INT, 0, 11,
MPI_COMM_WORLD);
MPI_Recv(&main_grid[0], DIM*DIM, MPI_INT, 0, 10,
MPI_COMM_WORLD, &status);
        } else {
for (inti=0; i<nrows*DIM; i++) {
main_grid[i] =changed_subgrid[i];
            }

for (intk=1; k<num_procs; k++) {
MPI_Recv(&main_grid[DIM* (DIM/num_procs) *k],
nrows*DIM, MPI_INT, k, 11, MPI_COMM_WORLD, &status);
            }
```

```
for (inti=1; i<num_procs; i++) {
MPI_Send(main_grid, DIM*DIM, MPI_INT, i, 10,
MPI_COMM_WORLD);
            }
        }

// Output the updated grid state
if (ID==0) {
doubleend=MPI_Wtime();
printf("Matrix DIM: %d\n", DIM);
printf("Kernel DIM: %d", KERNEL_DIM);
printf("\nConvolution Output: \n");
for (j=0; j<GRID_WIDTH; j++) {
if (j%DIM==0) {
printf("\n");
                }
printf("%d  ", main_grid[j]);
            }
printf("\n");

printf("Execution Time: %f\n", end-start_time);
        }
    }

if (num_procs>=2) {
free(pad_row_upper);
free(pad_row_lower);
    }

MPI_Finalize();
}
```

Output:

```
PS D:\Assignments> mpiexec -n 2 .\Q1.exe 10 8
Matrix DIM: 10
Kernel DIM: 8
Convolution Output:

16  20  24  28  28  28  28  24  20  16
20  25  30  35  35  35  35  30  25  20
24  30  36  42  42  42  42  36  30  24
28  35  42  49  49  49  49  42  35  28
28  35  42  49  49  49  49  42  35  28
28  35  42  49  49  49  49  42  35  28
28  35  42  49  49  49  49  42  35  28
24  30  36  42  42  42  42  36  30  24
20  25  30  35  35  35  35  30  25  20
16  20  24  28  28  28  28  24  20  16
Execution Time: 0.000978
```

For 500*500:

```
PS D:\Assignments> mpiexec -n 1 .\Q1.exe 500 8
Execution Time: 0.033009
PS D:\Assignments> mpiexec -n 2 .\Q1.exe 500 8
Execution Time: 0.017275
PS D:\Assignments> mpiexec -n 4 .\Q1.exe 500 8
Execution Time: 0.014928
```

```
PS D:\Assignments> mpiexec -n 20 .\Q1.exe 500 8
Execution Time: 0.023533
```

p	Time
1	0.033

2	0.0173
4	0.0149
20	0.0235

Conclusion:
Firstly with increasing p , execution time decreases then after saturation point it increases again.

2. Implement dot product using MPI. Use different number of processes and analyze the performance.

Code:

```
#include<math.h>
#include<mpi.h>
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<unistd.h>

#defineNELMS100000
#defineMASTER0
#defineMAXPROCS16

intdot_product();
voidinit_lst();
voidprint_lst();

intmain() {
inti, n, vector_x[NELMS], vector_y[NELMS];
```

```
int prod, sidx, eidx, size;
intpid, nprocs, rank;
doublestime, etime;
MPI_Status status;
MPI_Comm world;

    n =100000;
if (n > NELMS) {
printf("n=%d> N=%d\n", n, NELMS);
exit(1);
    }

MPI_Init(NULL, NULL);
    world = MPI_COMM_WORLD;
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &pid);

int portion = n /nprocs;
sidx=pid* portion;
eidx=sidx+ portion;
init_lst(vector_x, n);
init_lst(vector_y, n);

inttmp_prod[nprocs];
for (i=0; i<nprocs; i++)
tmp_prod[i] =0;

stime=MPI_Wtime();

if (pid== MASTER) {
        prod =dot_product(sidx, eidx, vector_x, vector_y,
n);
```

```
for (i=1; i<nprocs; i++)
MPI_Recv(&tmp_prod[i-1], 1, MPI_INT, i, 123,
MPI_COMM_WORLD, &status);
    } else {
        prod =dot_product(sidx, eidx, vector_x, vector_y,
n);
MPI_Send(&prod, 1, MPI_INT, MASTER, 123, MPI_COMM_WORLD);
    }

if (pid== MASTER) {
for (i=0; i<nprocs; i++)
            prod +=tmp_prod[i];
    }

etime=MPI_Wtime();

if (pid== MASTER) {
// print_lst(vector_x,n);
// print_lst(vector_y,n);
printf("pid=%d: final prod=%d\n", pid, prod);
printf("pid=%d: elapsed=%f\n", pid, etime-stime);
    }
MPI_Finalize();
}

intdot_product(int s,int e,int x[],int y[],int n) {
inti, prod =0;

for (i= s; i< e; i++)
        prod = prod +x[i] *y[i];

return prod;
```

```
}

voidinit_lst(int*l,int n) {
inti;
for (i=0; i< n; i++)
*l++=i;
}
voidprint_lst(int l[],int n) {
inti;

for (i=0; i< n; i++) {
printf("%d ", l[i]);
    }
printf("\n");
}
```

Output:

```
PS D:\Assignments> mpiexec -n 2 .\Q2.exe
pid=0: final prod=216474736
pid=0: elapsed=0.000839
PS D:\Assignments> mpiexec -n 4 .\Q2.exe
pid=0: final prod=216474736
pid=0: elapsed=0.001077
PS D:\Assignments> mpiexec -n 8 .\Q2.exe
pid=0: final prod=216474736
pid=0: elapsed=0.002098
PS D:\Assignments> mpiexec -n 16 .\Q2.exe
pid=0: final prod=216474736
pid=0: elapsed=0.003044
PS D:\Assignments>
```

p	time
1	0.00025
2	0.00083
4	0.00072
8	0.00210
16	0.00304

Conclusion: The increase or decrease in time is not regular it is increasing and decreasing.

3. Implement Prefix sum using MPI. Use different number of processes and analyze the performance.
   Code:

```
#include"mpi.h"
#include<math.h>
#include<stdio.h>
#include<stdlib.h>

intmain(intargc,char*argv[]) {
intmy_rank;    /* rank of process */
intp;          /* number of processes */
MPI_Statusstatus; /* return status for receive */
intvalue;

    /* start up MPI */
MPI_Init(&argc, &argv);
```

```
    /* find out process rank */
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* find out number of processes */
MPI_Comm_size(MPI_COMM_WORLD, &p);
intprefix_arr[p];

    /* getting input and scatter values */
if (my_rank==0) {
inti;
for (i=0; i<p; ++i) {
prefix_arr[i] =i+1;
        }
    }

doublestart=MPI_Wtime();

// all call scatter
MPI_Scatter(prefix_arr, 1, MPI_INT, &value, 1,
MPI_INT, 0, MPI_COMM_WORLD);

    /*
    prefix sum:
        repeat log n times
        each time, if we are the chosen one, we
receve a value from someone and add to ours
        otherwise, we send to the chosen one
    */
inti;
intlogn=log2(p);
for (i=0; i<=logn; i++) {
```

```
intlower_bound=pow(2, i);
intupper_bound=p-lower_bound;
if (upper_bound<lower_bound) {
upper_bound=lower_bound;
        }

if (my_rank<lower_bound) {
intsend= (int)(my_rank+pow(2, i));
if (send>=p)
continue;

printf("%d sending to %d\n", my_rank,
(int)(my_rank+pow(2, i)));
MPI_Send(&value, 1, MPI_INT, (int)(my_rank+pow(2,
i)), 0, MPI_COMM_WORLD);
        } elseif (my_rank>=upper_bound) {
intrecv= (int)(my_rank-pow(2, i));
if (recv>=p)
continue;

intrecv_value;
printf("%dreceving..\n", my_rank);
MPI_Recv(&recv_value, 1, MPI_INT, (my_rank-pow(2,
i)), 0, MPI_COMM_WORLD, &status);
value+=recv_value;
        } else {
intsend= (int)(my_rank+pow(2, i));
intrecv= (int)(my_rank-pow(2, i));
if (send>=p||recv>=p)
continue;
```

```
printf("%d sending to %d\n", my_rank,
(int)(my_rank+pow(2, i)));
MPI_Send(&value, 1, MPI_INT, (int)(my_rank+pow(2,
i)), 0, MPI_COMM_WORLD);

printf("%dreceving..\n", my_rank);
intrecv_value;
MPI_Statusstatus;
MPI_Recv(&recv_value, 1, MPI_INT, (my_rank-pow(2,
i)), 0, MPI_COMM_WORLD, &status);
value+=recv_value;
        }
    }
// after algorithm, each processor hols its own
prefix sum
// we gather at rank
intgather[p];
MPI_Gather(&value, 1, MPI_INT, gather, 1, MPI_INT, 0,
MPI_COMM_WORLD);

if (my_rank==0) {
doubleend=MPI_Wtime();

printf("Execution Time: %f\n", end-start);
    }

    /* shut down MPI */
MPI_Finalize();

return0;
}
```

Output:

```
PS D:\Assignments> mpiexec -n 4 .\Q3.exe
3 receving..
3 receving..
1 sending to 2
1 receving..
1 sending to 3
2 sending to 3
2 receving..
2 receving..
0 sending to 1
0 sending to 2
Execution Time: 0.001033
PS D:\Assignments> mpiexec -n 8 .\Q3.exe
1 sending to 2
1 receving..
1 sending to 3
1 sending to 5
7 receving..
7 receving..
7 receving..
2 sending to 3
2 receving..
2 sending to 4
2 receving..
2 sending to 6
0 sending to 1
0 sending to 2
0 sending to 4
Execution Time: 0.001867
5 sending to 6
5 receving..
5 sending to 7
5 receving..
5 receving..
3 sending to 4
3 receving..
3 sending to 5
3 receving..
3 sending to 7
6 sending to 7
```

p	time
2	0.0006
4	0.0010
8	0.0018
16	0.0033

Conclusion: With increasing p value time is gradually increasing.