

Appendix

Introduction to parallel programming with MPI

My first MPI program



C

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

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

    if (rank == 0)
        printf("Number of processes is %i\n",
              size);
    MPI_Barrier(MPI_COMM_WORLD);

    printf("Hello, I'm rank %i!\n", rank);
    MPI_Finalize();
}
```

Fortran

```
PROGRAM hello
    IMPLICIT NONE

    INCLUDE 'mpif.h'

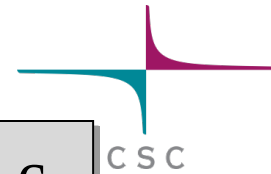
    INTEGER :: rank, size, ierror
    CALL MPI_INIT(ierror)
    CALL MPI_COMM_SIZE(MPI_COMM_WORLD,
size, ierror)
    CALL MPI_COMM_RANK(MPI_COMM_WORLD,
rank, ierror)

    IF (rank == 0) THEN
        WRITE(*,*) 'Number of processes
is', size
    END IF
    CALL MPI_BARRIER(MPI_COMM_WORLD,
ierror)

    WRITE(*,*) 'Hello, I am rank ', rank
    CALL MPI_FINALIZE(ierror)

END PROGRAM hello
```

Point-to-point case study 1: Parallel sum



Sum.c

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

int main(int argc, char *argv[]){
    int i,rank,nprocs, N;
    double *array;
    double sum,psum;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    N=100;
    if(rank==0){
        array=malloc(sizeof(double)*N);
        for(i=0;i<N;i++) array[i]=1.0;
        MPI_Send(array+N/2, N/2, MPI_DOUBLE, 1, 10, MPI_COMM_WORLD);
    }
    else{
        array=malloc(sizeof(double)*N);
        MPI_Recv(array, N, MPI_DOUBLE, 0, 10, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    psum=0;
    for(i=0;i<N/2;i++){
        psum+=array[i];
    }

    if(rank==0) {
        MPI_Recv(&sum, 1, MPI_DOUBLE, 1, 11, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        sum+=psum;
        printf("Sum is %g, partial sum %g\n", sum, psum);
    }
    else {
        MPI_Send(&psum, 1, MPI_DOUBLE, 0, 11, MPI_COMM_WORLD);
    }
    MPI_Finalize();
}
```

Point-to-point case study 1: Parallel sum



Version where the size of the received buffer is checked

```
MPI_Status status;
...
if(rank==0){
    Nloc=N/3;
    array=malloc(sizeof(double)*N);
    for(i=0;i<N;i++) array[i]=1.0;
    MPI_Send(array+Nloc,N-Nloc,MPI_DOUBLE,1,10,MPI_COMM_WORLD);
}
else{
    array=malloc(sizeof(double)*N);
    MPI_Recv(array,N,MPI_DOUBLE,0,10,MPI_COMM_WORLD,&status);
    MPI_Get_count(&status,MPI_DOUBLE,&Nloc);
}
printf("rank = %d Nloc =%d\n",rank,Nloc);
psum=0;
for(i=0;i<Nloc;i++){
    psum+=array[i];
}
....
```

Point-to-point case study 2: Jacobi

Some functions used by all three variants of this case study (jacobi_common.c)

```
void compute(double **V,double **Vn, double **beta,int ystart,int yend,int M){
    int i,j;
    for(i=ystart;i<=yend;i++){
        for(j=1;j<M-1;j++){
            Vn[i][j]=(V[i-1][j]+V[i+1][j]+V[i][j-1]+V[i][j+1]-beta[i][j])*0.25;
        }
    }
}

void init(double **V,double **Vn, double **beta,int N,int M){
    int i,j;
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    for(i=0;i<N+2;i++){
        for(j=0;j<M;j++){
            V[i][j]=0;
            Vn[i][j]=0;
            beta[i][j]=j%3-i%5;
        }
    }
}
```

Point-to-point case study 2: Jacobi

Version with MPI_Send and MPI_Recv: Page 1/2 (jacobi.c)

```
ngbr_up=rank-1;
ngbr_down=rank+1;

if(ngbr_up<0) ngbr_up=MPI_PROC_NULL;
if(ngbr_down>=nprocs) ngbr_down=MPI_PROC_NULL;

//size of local domain
N=N_global/nprocs;
M=M_global;

allocate_2Darray(&beta,N+2,M);
allocate_2Darray(&V ,N+2,M);
allocate_2Darray(&Vn ,N+2,M);

init(V,Vn,beta,N,M);

for(i=0;i<max_iterations;i++){
    if(rank%2==0){
        MPI_Send(V[1],M,MPI_DOUBLE,ngbr_up,10,MPI_COMM_WORLD);
        MPI_Recv(V[0],M,MPI_DOUBLE,ngbr_up,11,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        MPI_Send(V[N],M,MPI_DOUBLE,ngbr_down,12,MPI_COMM_WORLD);
        MPI_Recv(V[N+1],M,MPI_DOUBLE,ngbr_down,13,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }
}
...
```

Point-to-point case study 2: Jacobi

Version with MPI_Send and MPI_Recv: Page 2/2 (jacobi.c)

```
...
    if(rank%2==1){
        MPI_Recv(V[N+1], M, MPI_DOUBLE, ngrbr_down, 10, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Send(V[N], M, MPI_DOUBLE, ngrbr_down, 11, MPI_COMM_WORLD);
        MPI_Recv(V[0], M, MPI_DOUBLE, ngrbr_up, 12, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Send(V[1], M, MPI_DOUBLE, ngrbr_up, 13, MPI_COMM_WORLD);
    }

    compute(V, Vn, beta, 1, N, M);
    //swap V and Vn
    SWAP(V, Vn, dpptemp);
}
```

Point-to-point case study 2: Jacobi

Version with MPI_Sendrecv (jacobi_sendrecv.c).

```
for(i=0;i<max_iterations;i++){
    MPI_Sendrecv(V[1], M, MPI_DOUBLE, ngrbr_up, 10,
                 V[N+1], M, MPI_DOUBLE, ngrbr_down, 10, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv(V[N], M, MPI_DOUBLE, ngrbr_down, 11,
                 V[0], M, MPI_DOUBLE, ngrbr_up, 11, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    compute(V, Vn, beta, 1, N, M);
    //swap V and Vn
    SWAP(V, Vn, dpptemp);
}
```

Point-to-point case study 2: Jacobi

Version with MPI_Isend & MPI_Irecv (jacobi_nonblock.c).

```
MPI_Request requests[4];
...
for(i=0;i<max_iterations;i++){
    MPI_Irecv(V[N+1],M,MPI_DOUBLE,ngbr_down,10,MPI_COMM_WORLD,&requests[0]);
    MPI_Irecv(V[0],M,MPI_DOUBLE,ngbr_up,11,MPI_COMM_WORLD,&requests[1]);
    MPI_Isend(V[1],M,MPI_DOUBLE,ngbr_up,10,MPI_COMM_WORLD,&requests[2]);
    MPI_Isend(V[N],M,MPI_DOUBLE,ngbr_down,11,MPI_COMM_WORLD,&requests[3]);
    //compute inner parts
        compute(V,Vn,beta,3,N-2,M);
    //wait for communication to finish
    MPI_Waitall(4,requests,MPI_STATUSES_IGNORE);
    //compute borders
    compute(V,Vn,beta,1,2,M);
    compute(V,Vn,beta,N-1,N,M);
    //swap V and Vn
    SWAP(V,Vn,dpptemp);
}
```