

# Point-to-Point Communication

Christofer Fabián Chávez Carazas

Universidad Nacional de San Agustín

Algoritmos Paralelos

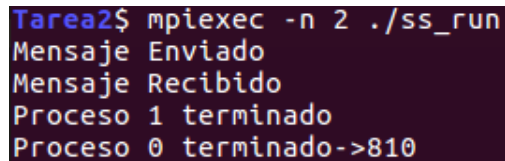
5 de abril de 2017

## 1. Blocking Synchronous Send

El primer proceso envía la señal listo para enviar y se bloquea hasta que el segundo proceso envíe una señal de que está listo para recibir. En ese momento el primer proceso envía el mensaje. El código se muestra a continuación y el resultado se muestra en la figura 1

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

int main(){
    srand(time(NULL));
    int id;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    if(id != 0){
        int random = rand() % 1000;
        int i = 0;
        printf("Mensaje Enviado\n");
        MPI_Ssend(&random, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        i += 10;
        i -= 5;
        printf("Proceso 1 terminado\n");
    }
    else{
        int res;
        MPI_Recv(&res, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Mensaje Recibido\n");
        res += 10;
        res -= 5;
        printf("Proceso 0 terminado->%d\n", res);
    }
    MPI_Finalize();
    return 0;
}
```



```
Tarea2$ mpiexec -n 2 ./ss_run
Mensaje Enviado
Mensaje Recibido
Proceso 1 terminado
Proceso 0 terminado->810
```

Figura 1: Resultados del Ssend

```

Tarea2$ mpiexec -n 2 ./rs_run
Mensaje Enviado
Proceso 1 terminado
Mensaje Recibido
Proceso 0 terminado->859

```

Figura 2: Resultados del Rsend

## 2. Blocking Ready Send

El segundo proceso envia la señal de listo para recibir y se bloquea hasta que el primer proceso envíe el mensaje. El código se muestra a continuación y los resultados en la figura 2

```

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

int main(){
    srand(time(NULL));
    int id;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    if(id != 0){
        int random = rand() % 1000;
        int i = 0;
        printf("Mensaje Enviado\n");
        MPI_Rsend(&random, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        i += 10;
        i -= 5;
        printf("Proceso 1 terminado\n");
    }
    else{
        int res;
        MPI_Recv(&res, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Mensaje Recibido\n");
        res += 10;
        res -= 5;
        printf("Proceso 0 terminado->%d\n", res);
    }
    MPI_Finalize();
    return 0;
}

```

## 3. Blocking Buffered Send

El usuario crea un buffer en donde el mensaje del primer proceso es almacenado; el primer proceso no se bloquea. Cuando el segundo proceso envia la señal de listo para recibir, el buffer transmite el mensaje. El código se muestra a continuación y los resultados en la figura 3

```

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

int main(){
    srand(time(NULL));
    int id;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    if(id != 0){
        int random = rand() % 1000;
        int i = 0;
        void * buffer;
        int buffsize = sizeof(int) + (2 * MPI_BSEND_OVERHEAD);
    }
}

```

```
Tarea2$ mpiexec -n 2 ./bs_run
Mensaje Enviado
Proceso 1 terminado
Mensaje Recibido
Proceso 0 terminado->510
```

Figura 3: Resultados del Bsend

```
buffer = (void *) malloc(buffsize*sizeof(void));
MPI_Buffer_attach(buffer, buffsize);
printf("Mensaje Enviado\n");
MPI_Bsend(&random, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
i += 10;
i -= 5;
printf("Proceso 1 terminado\n");
}
else{
int res;
MPI_Recv(&res, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
printf("Mensaje Recibido\n");
res += 10;
res -= 5;
printf("Proceso 0 terminado->%d\n", res);
}
MPI_Finalize();
return 0;
}
```

## 4. Blocking Standard Send

El MPI tiene un buffer interno con un tamaño fijo. Si el tamaño del mensaje es menor que el tamaño del buffer, el Standard Send se comporta como un Buffered Send; pero si el tamaño del mensaje supera el tamaño del buffer, el Standard Send se comporta como un Synchronous Send. El código se muestra a continuación y los resultados en la figura 4

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

int main(){
srand(time(NULL));
int id;
MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &id);
if(id != 0){
int random = rand() % 1000;
int i = 0;
printf("Mensaje Enviado\n");
MPI_Send(&random, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
i += 10;
i -= 5;
printf("Proceso 1 terminado\n");
}
else{
int res;
MPI_Recv(&res, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
printf("Mensaje Recibido\n");
res += 10;
res -= 5;
printf("Proceso 0 terminado->%d\n", res);
}
MPI_Finalize();
return 0;
}
```

```
Tarea2$ mpiexec -n 2 ./sts1_run  
Mensaje Enviado  
Proceso 1 terminado  
Mensaje Recibido  
Proceso 0 terminado->114
```

Figura 4: Resultados del Send