**Alumno** : Walter J. Felipe Tolentino
**Código** : 20172714F

# 1. Comunicación : blocking-comm vs nonblocking-comm

### Código fuente :

*blocking-comm.cpp*
*nonblocking-comm.cpp*

**Nota :** Se ha usado *MPI_Wait(&request, &status)* en la *comunicación* ***no bloqueante*** para sincronizar el recibo de la data, sino ocurre que posteriormente se calcula promedios con data <<*nula*>>. Pero aún así el tiempo de ejecución es menor que la *comunicación* ***bloqueante***, cómo se observa en los resultados que se muestran posteriormente.

### Comando para compilar y ejecutar :

*./run.sh*

**Nota :** *run.sh* hace todo el trabajo de compilado y ejecución iterando la cantidad de procesos **p** = 1,2,4,8 y el tipo de comunicación (**c**) . En cada combinación de ***p*** *y* ***c*** se hace **10** iteraciones para encontrar resultados estadísticamente más precisos.

### Resultados :

*results.txt*
**Formato :** *([PROMEDIO TOTAL], [TIEMPO])*

**Nota** : El tiempo considerado es la suma de las comunicaciones entre procesos y los respectivos cálculos en cada uno de ellos.

Analizando estadísticamente (promedio de tiempos) el contenido de *results.txt* se encontró los siguientes resultados :

| N° de procesos | Tiempo promedio de blocking-comm (s) | Tiempo promedio de nonblocking-comm (s) |
|---|---|---|
| 1 | 0.0034444332 | 0.0034079552 |
| 2 | 0.0030674934 | 0.0043859482 |
| 4 | 0.0863683224 | 0.0652427673 |
| 8 | 0.2196071148 | 0.0655798912 |
| **10** | **0.3150291443** | **0.0594246387** |
| **20** | **0.8429877758** | **0.0978584290** |

**Análisis :**

El tiempo de ejecución del mismo programa usando nonblocking-comm es menor que el tiempo usando blocking-comm, tal cual se puede observar en el cálculo de tiempo para **p = 4, 8, 10 y 20,** reduciendo el tiempo aproximadamente en un **75%**.

Debido que para el blocking-comm, el proceso emisor **espera** a que el receptor tenga la data para confirmar y validar el envío, luego de esto el proceso emisor continúa su flujo. Sin embargo, en el otro tipo de comunicación no ocurre eso, el emisor envía y no se preocupa de cuándo le llegue la data al receptor, así que el emisor continúa su flujo, es decir **no hay tiempo muerto.** Aunque en este último tipo de comunicación también se requiere sincronizar (*MPI_Wait*) puesto que se corre el riesgo de trabajar con datos nulos, sin embargo este tiempo de espera ocurre en el proceso receptor más no en el emisor, pues este ya envío y continua trabajando.

*Véase los resultados en results.txt o si desea realizar los propios ejecute run.sh*

## 2. Implementación de un DAG utilizando comunicación colectiva para el ejemplo mostrado en clase.

**Con Comunicación Colectiva** *dag-colective-comm.cpp*

```
14 void calculo(double i,double j,double k){
15     double a,x,d,suma; int p,rank;
16
17     MPI_Comm_size(MPI_COMM_WORLD,&p);
18     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
19
20     //fase 1
21     if(rank == 0) a=T1(i);
22     else if(rank == 1) a=T2(j);
23     else a=T3(k);
24     MPI_Reduce(&a, &suma, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);
25
26     //fase 2
27     if (rank==0) d = T4(suma);
28     MPI_Bcast(&d, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
29
30     //fase 3
31     if(rank == 0) x=T5(a/d);
32     else if(rank == 1) x=T6(a/d);
33     else x=T7(a/d);
34
35     //suma final
36     MPI_Reduce(&x, &suma, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);
37     if(rank==0) cout<<suma<<endl;
38 }
```

**Análisis**

En *dag-noncolective-comm.cpp* se cuentan **6** procesos de comunicación (*send -> recv*) proceso a proceso.
Por otro lado en *dag-colective-comm.cpp* se cuenta **3** procesos de comunicación colectiva en el siguiente orden

línea 24 : **MPI_Reduce**(&a, &**suma**, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);
description : suma el valor de la variable *a* de cada proceso y la almacena en la variable *suma.*

línea 28 : **MPI_Bcast**(&**d**, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
description : reparte el valor de la variable *d* a cada proceso.

línea 36 : **MPI_Reduce**(&x, &**suma**, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);
description : suma el valor de la variable *x* de cada proceso y la almacena en la variable *suma.*

**DAG (comunicación colectiva)**