CAGE – Grupo de Investigación

SC3 – Super Cómputo y Cálculo Científico – UIS

@sc3uis

Computación distribuida con MPI

Procesos, mensajes y comunicación entre nodos

Módulo 5 – Semillero CAGE

Luis Alejandro Torres Niño | sc3.operaciones@uis.edu.co

Objetivos del módulo

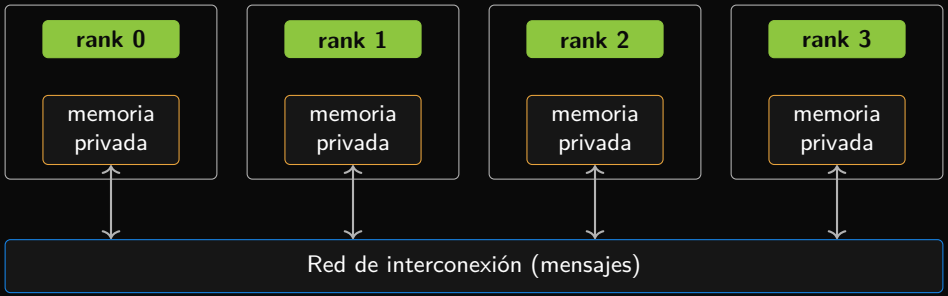
- ➔ Entender el modelo de **distributed memory** y por qué MPI es el estándar para usar varios nodos.
- ➔ Escribir programas MPI: ranks, communicators, `MPI_Send`/`MPI_Recv` y comunicación non-blocking.
- ➔ Usar **collective operations** (`Bcast`, `Scatter`, `Gather`, `Reduce`, `Allreduce`).
- ➔ Reconocer y evitar el ***deadlock***.
- ➔ Descomponer un dominio con **ghost cells** (*halo*) y hacer el *halo exchange*.
- ➔ Medir latencia, ancho de banda y escalabilidad; combinar MPI con OpenMP.
- ➔ Compilar y ejecutar con OpenMPI en el cluster usando SLURM.

Agenda

1. Distributed memory y el modelo MPI
2. Comunicación point-to-point y *deadlock*
3. Comunicación non-blocking
4. Collective operations
5. Domain decomposition y ghost cells
6. Rendimiento: latencia, ancho de banda y escalabilidad
7. MPI + OpenMP (híbrido)
8. MPI en el cluster con SLURM
9. Retos individuales y reto grupal

1
Distributed memory y el modelo MPI

Cada proceso tiene su propia memoria



- Un proceso **no puede leer** la memoria de otro: los datos viajan en **mensajes** explícitos.
- Los procesos pueden estar en el mismo nodo o en nodos distintos: **el programa es el mismo**.
- MPI = *Message Passing Interface*: un estándar (mpi-forum.org) con implementaciones OpenMPI, MPICH, etc.

SPMD: un programa, muchos ranks

- Lanzas **el mismo ejecutable** P veces: modelo **SPMD**.
- Cada copia tiene un **rank** (`rank`) de 0 a $P-1$ dentro de un **communicator**.
- `MPI_COMM_WORLD`: el communicator con todos los procesos.
- No confundir con **SIMD** (*Single Instruction, Multiple Data*): SPMD es un **modelo de programación**; SIMD es **paralelismo de hardware** (instrucciones vectoriales).
- El rank decide **qué parte** del trabajo hace cada proceso (`if (rank == 0)`…).

★ Esqueleto de todo programa MPI

```
MPI_Init(&argc, &argv);
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &
    rank);
MPI_Comm_size(MPI_COMM_WORLD, &
    size);
// ... trabajo ...
MPI_Finalize();
```

Compilar y ejecutar con OpenMPI

```
module avail openmpi mpi              # openmpi5/5.0.7, mpi/openmpi-gcc14/4.1.8,
    mpich ...
module load gnu14
module load mpi/openmpi-gcc14/4.1.8   # tambien: mpi/openmpi-gcc11/4.1.6,
    openmpi5/5.0.7

mpicxx -std=c++23 -O2 -Wall hola_mpi.cpp -o hola_mpi    # envoltorio de g++
mpirun -np 4 ./hola_mpi                                  # 4 procesos
```

- ➔ `mpicc`, `mpicxx`, `mpifort`: compiladores que añaden rutas y bibliotecas MPI.
- ➔ `mpicxx --showme` muestra la línea real de `g++`.

★ Cuidado con los módulos

Compila y ejecuta con **el mismo** módulo MPI. Mezclar OpenMPI y MPICH produce errores raros.

Hola MPI

```cpp
#include <mpi.h>
#include <cstdio>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank, size, len;
    char nodo[MPI_MAX_PROCESSOR_NAME];
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Get_processor_name(nodo, &len);
    std::printf("Hola desde el rank %d de %d en %s\n", rank, size, nodo);
    MPI_Finalize();
}
```

```
$ mpirun -np 4 ./hola_mpi
Hola desde el rank 3 de 4 en vm          # el orden de salida NO esta garantizado
Hola desde el rank 0 de 4 en vm
```

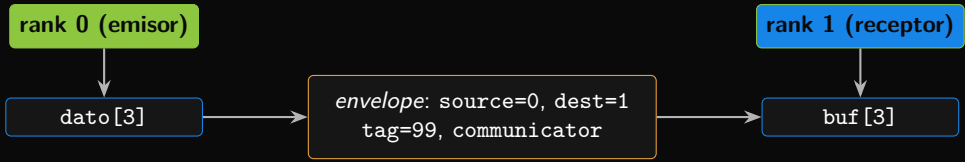
Reto individual 1: tus primeros ranks

■ Reto 1 – Hola desde varios nodos

1. Compila `hola_mpi.cpp` con el módulo `mpi/openmpi-gcc14/4.1.8`.
2. Ejecútalo con 4 procesos en un solo nodo con `sbatch` (script del final del módulo).
3. Repite con 2 nodos y 4 procesos por nodo (`--nodes=2 --ntasks-per-node=4`). ¿Qué nombres de nodo aparecen?
4. Haz que **solo el rank 0** imprima un encabezado con el número total de procesos y que los demás impriman su rank y nodo.

2
Comunicación point-to-point

Anatomía de un mensaje




```
MPI_Send(dato, 3, MPI_DOUBLE, /*dest*/ 1, /*tag*/ 99, MPI_COMM_WORLD);
MPI_Recv(buf,  3, MPI_DOUBLE, /*source*/ 0, /*tag*/ 99, MPI_COMM_WORLD, &estado);
```

Argumentos: **buffer**, **count**, **tipo**, **rank del otro**, **tag**, **communicator**. El *count* es en **elementos**, no bytes.

Tipos de datos MPI

Tipo en C/C++	Tipo MPI
`int`	`MPI_INT`
`long`	`MPI_LONG`
`float`	`MPI_FLOAT`
`double`	`MPI_DOUBLE`
`char`	`MPI_CHAR`
`unsigned long long`	`MPI_UNSIGNED_LONG_LONG`

★ Con std::vector

Pasa `v.data()` y `v.size()`: el vector es contiguo. Una **matriz por filas** en un `vector` también viaja de un solo envío. Para estructuras propias existen tipos derivados (`MPI_Type_create_struct`).

Enviar y recibir

```cpp
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

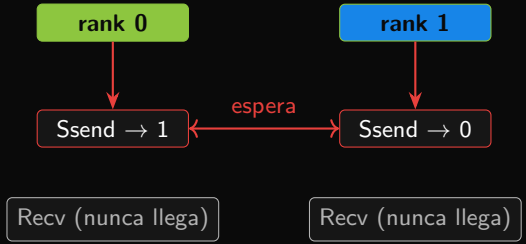
if (rank == 0) {
    double dato[3] = {1.5, 2.5, 3.5};
    MPI_Send(dato, 3, MPI_DOUBLE, 1, 99, MPI_COMM_WORLD);
} else if (rank == 1) {
    double buf[3];
    MPI_Status st;
    MPI_Recv(buf, 3, MPI_DOUBLE, 0, 99, MPI_COMM_WORLD, &st);
    std::printf("rank 1 recibio %.1f %.1f %.1f de %d\n",
                buf[0], buf[1], buf[2], st.MPI_SOURCE);
}
```

- ➔ `MPI_Send`/`MPI_Recv` son **blocking**: `MPI_Send` regresa cuando el buffer es seguro de reutilizar; `MPI_Recv`, cuando el mensaje ya está en el buffer.
- ➔ Receptor flexible: `MPI_ANY_SOURCE`, `MPI_ANY_TAG`; consulta `estado.MPI_SOURCE`.

Deadlock: ambos esperan a que el otro reciba

```c
// ambos ranks ejecutan lo mismo
int otro = 1 - rank, x = rank, y;
MPI_Ssend(&x, 1, MPI_INT, otro, 0,
          MPI_COMM_WORLD);   // espera
              al recv
MPI_Recv (&y, 1, MPI_INT, otro, 0,
          MPI_COMM_WORLD,
              MPI_STATUS_IGNORE);
```

`Ssend` termina solo cuando el receptor **empezó** a recibir. Con `Send` estándar a veces "funciona" si el mensaje es pequeño (se copia a un *buffer* interno), pero **no es seguro**: con mensajes grandes se bloquea.



Evitar el deadlock

```
// Solucion 1: MPI_Sendrecv combina envio y recepcion sin deadlock
MPI_Sendrecv(&x, 1, MPI_INT, otro, 0,
             &y, 1, MPI_INT, otro, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

// Solucion 2: romper la simetria (pares envian primero, impares reciben primero)
if (rank % 2 == 0) { MPI_Send(&x,1,MPI_INT,otro,0,MPI_COMM_WORLD); MPI_Recv(&y,1,
    MPI_INT,otro,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE); }
else               { MPI_Recv(&y,1,MPI_INT,otro,0,MPI_COMM_WORLD,
    MPI_STATUS_IGNORE); MPI_Send(&x,1,MPI_INT,otro,0,MPI_COMM_WORLD); }

// Solucion 3: comunicacion non-blocking (siguiente seccion)
```

❄ Regla

Cada `Send` necesita un `Recv` que **pueda ejecutarse** sin depender de que otro `Send` termine. Dibuja el orden de las llamadas de cada rank antes de programar.

Reto individual 2: el anillo

■ Reto 2 – Token ring

1. Con P procesos, el rank 0 envía el valor 0 al rank 1; cada rank suma su número de rank y lo pasa al siguiente; el último lo devuelve al 0.
2. El rank 0 imprime el resultado final y comprueba que vale $P(P-1)/2$.
3. Pruébalo con 2, 4, 8 y 16 procesos. ¿Qué pasa si todos hacen `Send` antes de `Recv` en el anillo?
4. Mide cuánto tarda una vuelta con `MPI_Wtime` y calcula el tiempo por salto.

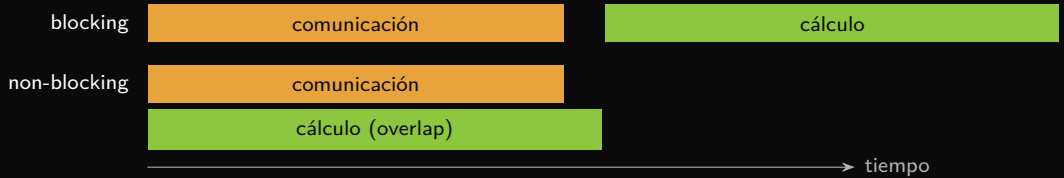
3
Comunicación non-blocking

Isend, Irecv y Wait

```
MPI_Request req[2];
MPI_Irecv(buf_in,  n, MPI_DOUBLE, vecino, 0, MPI_COMM_WORLD, &req[0]);   //
    regresa YA
MPI_Isend(buf_out, n, MPI_DOUBLE, vecino, 0, MPI_COMM_WORLD, &req[1]);

calcular_interior();                          // trabajo que NO usa buf_in ni
    modifica buf_out

MPI_Waitall(2, req, MPI_STATUSES_IGNORE); // ahora si: los datos estan listos
usar(buf_in);
```



Reglas de la comunicación non-blocking

- **No toques** el buffer de un `Isend`/`Irecv` hasta completar con `Wait`/`Waitall`/`Test`.
- `MPI_Wait(&req, &st)` espera una operación; `MPI_Waitall` espera varias; `MPI_Test` pregunta sin bloquear.
- Un `Isend` puede emparejarse con un `Recv` normal y viceversa.
- El overlap real depende de la implementación y la red: **mide** antes de afirmar que ganas tiempo.

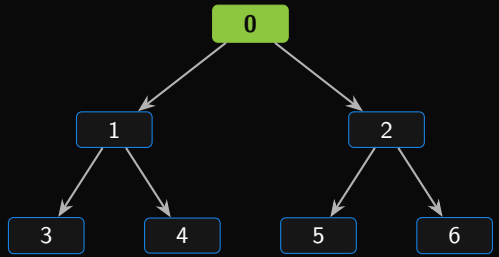
★ Patrón típico en HPC

1) Lanzar `Irecv`/`Isend` de las fronteras. 2) Calcular el **interior** (no depende de la frontera). 3) `Waitall`. 4) Calcular la **frontera**.

4
Collective operations

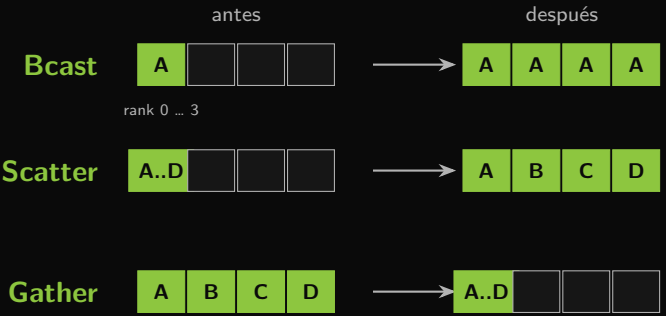
Qué son las collectives

- Las llaman **todos** los procesos del communicator, con los mismos argumentos de "qué" y "quién es la *root*".
- Más simples, claras y **más rápidas** que implementarlas con `Send`/`Recv`: la biblioteca usa algoritmos en árbol.
- No necesitan *tags*.



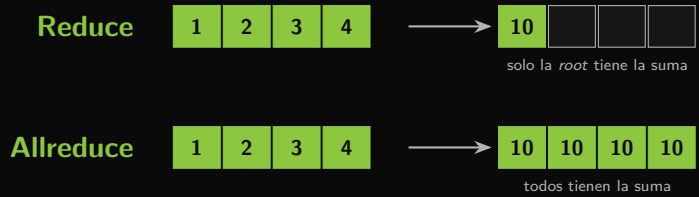
`Bcast` en árbol: $\lceil \log_2 P \rceil$ pasos (árbol binomial)

Bcast, Scatter y Gather



Cada celda es el dato de un rank (0, 1, 2, 3 de izquierda a derecha). `Scatter`: la *root* **reparte** un trozo a cada rank. `Gather`: la *root* **junta** un trozo de cada rank.

Reduce y Allreduce




```c
double local = calcular_parcial(), total = 0.0;
MPI_Reduce    (&local, &total, 1, MPI_DOUBLE, MPI_SUM, /*root*/ 0, MPI_COMM_WORLD)
    ;
MPI_Allreduce(&local, &total, 1, MPI_DOUBLE, MPI_SUM,            MPI_COMM_WORLD);
```

Operaciones: `MPI_SUM`, `MPI_PROD`, `MPI_MAX`, `MPI_MIN`, `MPI_LAND`… y `MPI_MAXLOC`/`MINLOC` (valor + posición).

Las collectives en un programa

```cpp
int n = 0;
if (rank == 0) n = 100;
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);                    // todos saben n

std::vector<int> todo;                                             // solo la root
    lo llena
if (rank == 0) { todo.resize(size * 2); for (int i = 0; i < size * 2; ++i) todo[i
    ] = i; }
int mio[2];
MPI_Scatter(todo.data(), 2, MPI_INT, mio, 2, MPI_INT, 0, MPI_COMM_WORLD);

int local = mio[0] + mio[1], total = 0, maximo;
MPI_Reduce   (&local, &total,  1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Allreduce(&local, &maximo, 1, MPI_INT, MPI_MAX,    MPI_COMM_WORLD);
// con 4 procesos: total = 28, maximo = 13
```

En `Scatter`/`Gather` el *count* es el que recibe/envía **cada** rank. Si el total no es divisible entre P, usa `MPI_Scatterv`/`MPI_Gatherv`.

Ejemplo: pi con MPI

```
const long N = 400000000;
const double h = 1.0 / N;
double t0 = MPI_Wtime();

double local = 0.0;
for (long i = rank; i < N; i += size) {          // reparto ciclico de
    iteraciones
    double x = (i + 0.5) * h;
    local += 4.0 / (1.0 + x * x);
}
double pi = 0.0;
MPI_Reduce(&local, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

if (rank == 0)
    std::printf("pi ~ %.10f  (%d procesos, %.3f s)\n", pi * h, size, MPI_Wtime()
        - t0);
```

Mismo problema que en el módulo 4 (aquí con $N = 4 \times 10^8$), pero cada **proceso** calcula su parte y `MPI_Reduce` suma los parciales.

Reto individual 3: pi y escalabilidad

■ Reto 3 – Speedup con MPI

1. Compila el programa de π con `mpicxx -O2` y ejecútalo con 1, 2, 4, 8 y 16 procesos en un nodo.
2. Repite con 16, 32 y 64 procesos en **varios nodos** (`--nodes=2` y luego `--nodes=4`, ajustando `--ntasks-per-node`).
3. Calcula speedup y eficiencia y grafícalos.
4. Compara con la versión OpenMP del módulo 4: ¿cuál escala más allá de un nodo? ¿Por qué?
5. *Extra:* reemplaza `Reduce` por `Allreduce` y mide la diferencia.

Reto individual 4: producto matriz–vector

■ Reto 4 – Distribuir y recolectar

1. El rank 0 crea una matriz A de $N \times N$ (por filas) y un vector x.
2. `Bcast` de x a todos; `Scatter` de las **filas** de A (cada rank recibe N/P filas).
3. Cada rank calcula su trozo de $y = Ax$; el rank 0 junta con `Gather`.
4. Verifica contra el cálculo secuencial. Prueba con N **no divisible** entre P usando `Scatterv`/`Gatherv`.
5. Mide cuánto tiempo es **cálculo** y cuánto **comunicación**. ¿Con qué N vale la pena paralelizar?

5
Domain decomposition y ghost cells

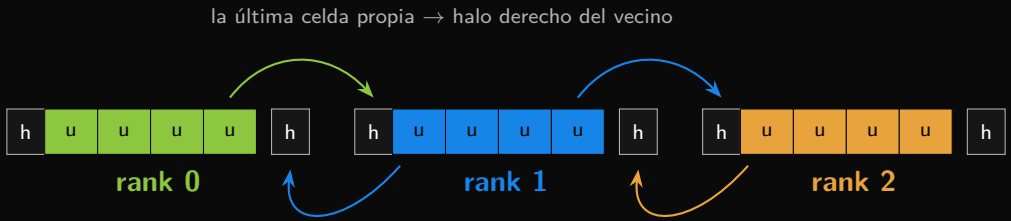
Repartir el dominio

grid 1D de $N = 16$ celdas, $P = 4$ ranks: cada uno posee $N/P = 4$



- **Cómputo local**, **datos locales**: cada proceso guarda solo su trozo (la memoria total crece con P).
- Para actualizar sus celdas de borde necesita el valor de la celda vecina, que vive en **otro** proceso.
- Solución: **ghost cells** (*halo*) que son copias de las fronteras del vecino, refrescadas en cada paso.

Halo exchange



`u` = celdas propias, `h` = halo. Cada paso: **1)** intercambiar halos, **2)** actualizar celdas propias.

Difusión 1D con halo

```
const int N = 1000, pasos = 2000, n = N / size;            // n celdas por rank
std::vector<double> u(n + 2, 0.0), un(n + 2, 0.0);         // +2: halo izq y der
if (rank == 0) u[0] = 100.0;                               // borde caliente

int izq = (rank > 0)        ? rank - 1 : MPI_PROC_NULL;    // PROC_NULL: no hace nada
int der = (rank < size - 1) ? rank + 1 : MPI_PROC_NULL;

for (int t = 0; t < pasos; ++t) {
    MPI_Sendrecv(&u[1],   1, MPI_DOUBLE, izq, 0,  &u[n+1], 1, MPI_DOUBLE, der, 0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
    MPI_Sendrecv(&u[n],   1, MPI_DOUBLE, der, 1,  &u[0],   1, MPI_DOUBLE, izq, 1, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
    if (rank == 0) u[0] = 100.0;
    for (int i = 1; i <= n; ++i) un[i] = u[i] + 0.4 * (u[i-1] - 2*u[i] + u[i+1]);
    for (int i = 1; i <= n; ++i) u[i] = un[i];
}
```

➔ `MPI_PROC_NULL` simplifica los extremos: enviar/recibir con él no hace nada.

➔ **Resultado verificado**: calor total $= 3141.987023$ con 1, 2 y 4 procesos (idéntico).

Reto individual 5: halo en una dimensión

■ Reto 5 – Difusión con Isend/Irecv

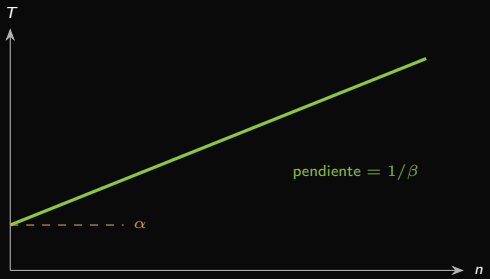
1. Parte de la difusión 1D con `Sendrecv` y compruébala con 1, 2, 4 y 8 procesos (el calor total debe coincidir).
2. Reescribe el intercambio con `Isend`/`Irecv`: calcula las celdas **interiores** mientras llegan los halos y luego las dos celdas de borde.
3. Mide el tiempo con $N = 10^7$ y 1000 pasos. ¿Hay overlap real?
4. Cuenta bytes enviados por paso y por rank; ¿cómo cambia al crecer P?

6

Rendimiento: latencia, ancho de banda y escalabilidad

El modelo de costo de un mensaje

$$T(n) \approx \alpha + \frac{n}{\beta}$$



- ➔ α: **latencia** (costo fijo por mensaje).
- ➔ β: **ancho de banda** (bytes/s).
- ➔ Mensajes pequeños: domina α; grandes: domina n/β.
- ➔ **Menos mensajes, más grandes** suele ser mejor que muchos pequeños.

Ping-pong: dos procesos se envían un mensaje de ida y vuelta; se promedian muchas repeticiones (Reto 6).

Ping-pong medido

Tamaño	Tiempo (una vía)	Ancho de banda
8 B	0.44 μs	18 MB/s
128 B	0.57 μs	223 MB/s
2 KB	1.02 μs	2.0 GB/s
32 KB	8.8 μs	3.7 GB/s
512 KB	62.9 μs	8.3 GB/s
8 MB	963 μs	8.7 GB/s

- Medido con OpenMPI 4.1.6, 2 procesos **en el mismo nodo** (shared memory) de una VM pequeña.
- Entre nodos la latencia es mucho mayor y el ancho de banda lo fija la red (consulta el tipo de red del cluster): **mide en** `guane`.
- La curva muestra la forma $\alpha + n/\beta$: tiempo casi plano y luego lineal.

Strong y weak scaling

Strong scaling

- Problema **fijo**, más procesos.
- Ideal: $T_p = T_1/p$.
- Se limita por Amdahl y por la comunicación, que pesa más cuando cada proceso tiene poco trabajo.

Weak scaling

- Trabajo **por proceso** fijo; el problema crece con p.
- Ideal: $T_p = T_1$ constante.
- Mide si el sistema aguanta problemas más grandes (más memoria total).

★ Relación cómputo/comunicación

En un dominio 1D con halo, cada rank calcula $\propto N/P$ celdas y comunica un tamaño constante: al subir P la comunicación **pesa cada vez más**. Por eso se mide en **ambos** regímenes y se reporta eficiencia.

Qué suele frenar a un programa MPI

- ➔ ***Load imbalance***: todos esperan al rank más lento en la siguiente collective o intercambio.
- ➔ ***Collectives* innecesarias**: un `Barrier` o `Allreduce` por paso cuando bastaba cada k pasos.
- ➔ **Muchos mensajes pequeños**: agrupa datos en un solo mensaje.
- ➔ **Cuello de botella en el rank 0**: leer/escribir todo en un solo proceso; usa E/S paralela cuando crezca el problema.
- ➔ **Mal *process placement***: procesos que se comunican mucho en nodos lejanos.

★ Diagnóstico rápido

Mide con `MPI_Wtime` por fase (cálculo, comunicación, E/S) en **cada rank** y reduce con `MPI_MAX` y `MPI_MIN`: si máximo y mínimo difieren mucho, hay desbalance.

Reto individual 6: mide tu red

■ Reto 6 – Ping-pong

1. Escribe un ping-pong entre los ranks 0 y 1 con tamaños de 1 byte a 64 MB (potencias de 2) y 200 repeticiones por tamaño.
2. Ejecútalo con los dos ranks en **el mismo nodo** y en **nodos distintos** (`--nodes=2 --ntasks-per-node=1`).
3. Grafica $T(n)$ y el ancho de banda; estima α y β de cada caso.
4. ¿A partir de qué tamaño el mensaje deja de estar dominado por la latencia?

7
MPI + OpenMP: programación híbrida

¿Por qué híbrido?



- **Un rank por nodo** (o por socket) y **un hilo por núcleo**: menos procesos, menos mensajes y menos memoria duplicada (halos).
- MPI entre nodos; OpenMP dentro (módulo 4).
- Menos ranks $\Rightarrow$ collectives más baratas.

Niveles de soporte de hilos

Nivel solicitado

`MPI_THREAD_SINGLE`

`MPI_THREAD_FUNNELED`

`MPI_THREAD_SERIALIZED`

`MPI_THREAD_MULTIPLE`

de menos a más permisivo: solo un hilo; solo el hilo principal llama a MPI; un hilo a la vez; cualquier hilo cuando quiera.

Compilar: `mpicxx -fopenmp`.

```
int req = MPI_THREAD_FUNNELED, prov;
MPI_Init_thread(&argc, &argv, req, &prov)
    ;
// comprueba que prov >= req
#pragma omp parallel
{
    // los hilos calculan; MPI solo
    // desde el hilo principal, fuera
    // de la region paralela
}
```

Script SLURM híbrido

```
#!/bin/bash
#SBATCH --job-name=hibrido
#SBATCH --partition=legacy
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=1          # 1 rank MPI por nodo
#SBATCH --cpus-per-task=8            # 8 hilos por rank
#SBATCH --time=00:10:00
#SBATCH --output=hib_%j.out

module load gnu14 mpi/openmpi-gcc14/4.1.8
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
export OMP_PROC_BIND=close OMP_PLACES=cores

mpirun -np $SLURM_NTASKS --map-by ppr:1:node:PE=$SLURM_CPUS_PER_TASK --report-
    bindings ./hibrido
```

`--report-bindings` muestra a qué núcleos queda ligado cada rank: úsalo para verificar que los hilos no se pisan.

8
MPI en el cluster con SLURM

Script SLURM para MPI

```
#!/bin/bash
#SBATCH --job-name=mpi_pi
#SBATCH --partition=legacy
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=8          # 16 procesos MPI en total
#SBATCH --time=00:10:00
#SBATCH --output=pi_%j.out

module load gnu14 mpi/openmpi-gcc14/4.1.8

mpirun -np $SLURM_NTASKS ./pi_mpi
```

- `mpirun` de OpenMPI lee de SLURM los nodos asignados; `-np` = total de procesos.
- Alternativa: `srun ./pi_mpi`. Funciona si SLURM tiene PMI/PMIx habilitado; comprueba con `srun --mpi=list`.
- Consulta los núcleos por nodo con `scontrol show node guane01` antes de elegir `--ntasks-per-node`.

Opciones útiles de mpirun

Opción	Para qué sirve
`-np N`	Número total de procesos.
`--map-by node`	Reparte los ranks **en turnos** entre los nodos.
`--map-by core`	Llena los núcleos de un nodo antes de pasar al siguiente.
`--bind-to core`	Fija cada rank a un núcleo (evita migraciones).
`--report-bindings`	Imprime dónde quedó cada rank.
`--oversubscribe`	Permite más ranks que núcleos (solo para pruebas).

Las opciones pueden variar entre OpenMPI 4.1 y 5.0: revisa `mpirun --help`.

Errores frecuentes

- ➔ **El programa se cuelga**: casi siempre un *deadlock* o una collective que no llaman todos los ranks. Imprime el rank antes y después de cada llamada.
- ➔ **Cantidad o tipo distintos** en emisor y receptor: truncamiento del mensaje (`MPI_ERR_TRUNCATE`).
- ➔ **Olvidar** `MPI_Finalize` o llamar a MPI antes de `MPI_Init`.
- ➔ **Mezclar módulos** (compilar con uno y ejecutar con otro).
- ➔ **Salida desordenada**: ordena tú; haz que el rank 0 imprima resultados o usa `--tag-output` (OpenMPI).
- ➔ **Leer un archivo en todos los ranks**: lee en el 0 y reparte, o usa E/S paralela.

★ Depurar con pocos procesos

Reproduce el error con 2 o 4 procesos antes de lanzar 100: `mpirun -np 4 ./prog`. Compila con `-g -fsanitize=address` para memoria.

9
Reto grupal

Reto grupal: difusión de calor 2D distribuida

★ Reto grupal – Equipos de 3 o 4 personas

Lleven al cluster multi-nodo la difusión de calor 2D (Jacobi) del módulo 4:

1. **Descomposición por bandas de filas**: cada rank tiene N/P filas más 2 filas de halo.
2. Intercambio de halos con `Sendrecv` (y luego con `Isend`/`Irecv` con overlap del cálculo interior).
3. Criterio de convergencia con `MPI_Allreduce(MPI_MAX)` sobre el error local.
4. Barridos de **strong** y **weak** scaling con 1 a 4 nodos; compárenlos con OpenMP.
5. *Extra:* versión híbrida (MPI + OpenMP) o descomposición 2D.

Esquema del reto grupal

```
// T tiene (n+2) filas x N columnas: filas 0 y n+1
     son halo
int arriba = rank > 0        ? rank - 1 :
     MPI_PROC_NULL;
int abajo  = rank < size - 1 ? rank + 1 :
     MPI_PROC_NULL;
for (int it = 0; it < maxit; ++it) {
    MPI_Sendrecv(&T[1*N],     N, MPI_DOUBLE, arriba,
         0,
                 &T[(n+1)*N], N, MPI_DOUBLE, abajo,
                      0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv(&T[n*N],     N, MPI_DOUBLE, abajo,
         1,
                 &T[0],       N, MPI_DOUBLE, arriba,
                      1,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    double err = jacobi_local(T, Tn, n, N);  //
         OpenMP aqui
    std::swap(T, Tn);
    double gerr;
    MPI_Allreduce(&err, &gerr, 1, MPI_DOUBLE, MPI_MAX
         ,
                  MPI_COMM_WORLD);
    if (gerr < tol) break;
}
```



Cada **fila** es contigua en memoria: una fila de halo viaja como un solo mensaje de N doubles.

Entregables y criterios

Entregables

- ➔ Código MPI (y híbrido, si lo hacen) con `Makefile`.
- ➔ Scripts `sbatch` de los barridos.
- ➔ Tablas y gráficas de strong y weak scaling.
- ➔ Informe corto: qué limita el escalado.

Criterios

- ➔ Mismo resultado que la versión secuencial, para cualquier P.
- ➔ Sin *deadlocks*; halos correctos en los extremos.
- ➔ Mediciones repetidas, con nodos exclusivos.
- ➔ Análisis con el modelo $\alpha + n/\beta$ y Amdahl.

Reutilización: usen su `minila` (módulo 3): distribuyan el producto de matrices por bandas de filas, con `Bcast` de B.

Referencias

- MPI Forum, *MPI: A Message-Passing Interface Standard*, versión 4.1 (2023): mpi-forum.org.
- Open MPI Project, documentación y página de `mpirun`: open-mpi.org.
- W. Gropp, E. Lusk, A. Skjellum, *Using MPI*, 3.ª ed., MIT Press, 2014.
- P. Pacheco y M. Malensek, *An Introduction to Parallel Programming*, 2.ª ed., Morgan Kaufmann, 2021.
- B. Barney, *Message Passing Interface (MPI)*, Lawrence Livermore National Laboratory: hpc-tutorials.llnl.gov/mpi.
- G. Amdahl, *Validity of the single processor approach…*, AFIPS Spring Joint Computer Conf., 1967.
- Material del semillero: módulos 1 (SLURM), 3 (C/C++) y 4 (OpenMP).

Cierre

★ MPI en una línea

Procesos con memoria separada que se comunican con mensajes: piensa en *ranks*, evita el *deadlock* y **mide la comunicación** antes de escalar a más nodos.

¡Gracias!

sc3.operaciones@uis.edu.co