1 – Conceptos básicos

1.1 – Concurrencia

Es la capacidad de ejecutar más de un proceso a la vez, sin necesidad de que un proceso determinado deba esperar a que finalice completamente uno anterior. Pueden ser totalmente independientes o haber interacciones entre ellos. Es un concepto de software.

1.2 – Paralelismo

Es una de las formas de lograr la concurrencia, mediante la ejecución de distintas tareas en diferentes procesadores en simultáneo. Es una característica del hardware.

1.3 – Multitarea

Es una forma de lograr la concurrencia en los sistemas con un único procesador, es decir, monoprocesador. Consiste en ejecutar porciones de cada tarea habilitada (slices) de manera alternada (context switch) en una única CPU. Imperceptible para el usuario.

1.4 – Sincronización

Refiere a la posesión de información acerca de otro proceso para poder coordinar actividades. Un primer caso es la sincronización **por exclusión mutua**, que consiste en asegurar que un único proceso tenga acceso a un recurso compartido en un instante de tiempo, o también que un único proceso se encuentre en una sección crítica a la vez.

El otro es la sincronización **por condición**, que consiste en bloquear la ejecución de un proceso hasta que se cumple una condición para asegurar un orden temporal.

Ejemplo que use ambos: productores y consumidores que utilicen un Buffer en memoria compartida. El acceso al Buffer debe realizarse con exclusión mutua, y luego con variables condición se puede bloquear procesos hasta que el Buffer esté libre o lleno.

1.5 – Interferencia

Es la situación donde un proceso realiza una acción que invalida las suposiciones hechas por otro proceso. Por ejemplo, el valor de una variable compartida.

1.6 – Granularidad

Es la relación entre el trabajo de cómputo (útil) y el de comunicación. Depende del tamaño de las tareas y su dependencia con las demás. Si hay poca necesidad de sincronización, decimos que es de **grano grueso**, caso contrario es de **grano fino**.

1.7 – Fairness

Es una propiedad deseable que consiste en lograr un equilibrio en el acceso a los recursos compartidos por todos los procesos. Trata de garantizar que todos los procesos tengan chance de avanzar, sin importar las acciones del resto.

1.8 – Deadlock 💀

Es una situación indeseable causada por un error de programación, que consiste en que 2 o más procesos permanecen bloqueados debido a que ambos se encuentran esperando a que el otro libere un recurso compartido. Su ausencia es necesaria.

2 – Atomicidad

2.1 – Acción atómica

Es una instrucción que realiza una transformación **indivisible** de un estado a otro de un programa concurrente: no hay estados intermedios visibles para otros procesos.

Cada sentencia puede estar compuesta por 1 o más acciones atómicas. Al ejecutar el programa concurrente, se intercalan las acciones atómicas ejecutadas por cada proceso individual, formando una **historia** (trace) particular que puede ser válida o no.

Puede ser de **grano fino**, que debe implementarse por hardware…

- Ejemplo bueno: asignar el valor 3 a la variable “X” (Store 3, PosMemX).
- Ejemplo malo: asignar A = B, ambos en memoria (requiere Load y Store).

2.2 – Referencia crítica

Sucede cuando una expresión hace referencia a una variable que es modificada por otro proceso. Si no tiene referencias críticas, la ejecución es atómica.

2.3 – Tipos de Fairness

Es una característica que se aplica a las políticas de scheduling.

- **Incondicional:** toda acción atómica incondicional y elegible será ejecutada en algún momento (elegible es la próxima de un proceso). Ejemplo: RR en mono.
- **Débil:** ídem y además toda acción atómica condicional y elegible será ejecutada en algún momento, suponiendo que la condición se vuelve verdadera y permanece verdadera hasta que es vista por el proceso que ejecuta la acción atómica.
- **Fuerte:** ídem anterior, pero la condición se asigna en true constantemente.

3 – Problema de la Sección Crítica

- Comenzado el 13 de jul. de 22, de clase n°02, filminas 25 a 57.

3.1 – Introducción

Se debe implementar el concepto de “acciones atómicas” en software.

3.2 – Propiedades a cumplir

Los protocolos de entrada y salida de una sección crítica deben cumplir:

- ✓ Exclusión mutua: 0 o 1 procesos pueden encontrarse en la sección crítica.
- ✓ Eventual entrada: un proceso que intenta acceder tiene posibilidades de hacerlo.
- ✓ Ausencia de Deadlock: si más de un proceso intenta acceder a la sección crítica, al menos uno tendrá éxito. No existe posibilidad de bloqueo permanente.
- ✓ Ausencia de demora innecesaria: si un proceso intenta acceder a la sección crítica, y los demás están en sus SNC o terminaron, no está impedido de entrar.

3.3 – Algoritmo Ticket 

Es una solución con busy-waiting. Se tienen 3 variables compartidas: un vector numérico de turnos, el próximo y el número que le corresponde al siguiente proceso que arriba (a modo de tickeadora). Cada proceso se asigna el número en su posición del vector, y lo incrementa para evitar que se repita. Luego, para ingresar a la sección crítica se espera hasta que “próximo” sea igual al turno asignado. Antes de salir, incrementa próximo.

3.4 – Algoritmo Bakery 

Se tiene una única variable compartida: un vector numérico de turnos. Cada proceso, en primer lugar, recorre el vector de turnos para hallar el máximo y autoasignarse el valor siguiente a dicho máximo. Luego, para poder ingresar a la sección crítica se espera hasta que su número de turno sea el menor de los que esperan. Antes de salir de la sección crítica, reinicia su número de turno a cero. Los procesos se chequean entre ellos.

3.5 – Spin Lock 

Se tiene una única variable compartida booleana “lock”, inicializada en false. La solución consiste en que cada proceso debe esperar a que la variable “lock” se encuentre en false para poder establecerla en *true* e ingresar a la sección crítica. Al momento de salir de la misma, se debe establecer *lock* en *false*, para que haya ausencia de deadlock.

4 – Semáforos

- Comenzado el 14 de jul. de 22, de clase n°03, filminas 4 a 34.

4.1 – Introducción

Son instancias de un tipo de dato abstracto que tiene un contador interno que toma valores enteros no negativos. Brinda 2 operaciones de ejecución atómica: P y V.

P se utiliza para demorar un proceso hasta que ocurra un evento, decrementando el contador (se demora si vale 0). V representa la ocurrencia de un evento (incrementa).

4.2 – Sincronización

Respecto a la exclusión mutua, se utiliza un semáforo *mutex* inicializado en 1, para acceder a la sección critica se realiza un **P**, y al momento de salir se realiza un **V**. En caso de exclusión mutua selectiva, es decir, cuando los procesos requieren acceder a más de un recurso compartido, se hace un **P** para cada uno. Recordar caso de los filósofos.

La sincronización por condición se realiza con semáforos para cada evento, que pueden inicializarse en 0 o 1. Por ejemplo, para productores y consumidores, se puede tener un semáforo de “lleno” inicializado en 0 y “vacio” inicializado en 1.

5 – Monitores

5.1 – Introducción

Son un mecanismo de abstracción de datos, que encapsulan las representaciones de los recursos y brindan un conjunto de operaciones para manipular los mismos.

Contienen variables permanentes (privadas) y *procedures* (públicos). Éstos últimos pueden acceder a las variables permanentes, sus locales y los parámetros.

5.2 – Sincronización

Respecto a la exclusión mutua, se encuentra implícita porque los *procedures* de un mismo monitor no se pueden ejecutar concurrentemente.

La sincronización por condición es explicita con variables condición. Se declaran con el tipo "cond", y representan una cola de procesos demorados. Sus operaciones son:

- Wait: el proceso se demora, se agrega al final de la cola de la variable condición y deja el acceso exclusivo al monitor (otros pueden entrar a *procedures*).
- Signal: se despierta al proceso que se encuentra primero en la cola de la variable condición, y lo saca de la misma. Si no hay ninguno, no hay error. Sin embargo, el proceso despertado podrá continuar cuando readquiera el acceso al monitor.
    - Signal and continue: el proceso que realiza el signal **continúa** usando el monitor, y el despertado pasa a competir por el acceso al monitor.
    - Signal and wait: el proceso que realiza el signal **pasa a competir** por el acceso al monitor, y el despertado continúa su ejecución (sin competir).
- Signal_all: despierta y saca a todos los procesos de la cola.

5.3 – Diferencia con Semáforos

Los semáforos poseen un contador interno mientras las variables condición poseen una cola interna de procesos demorados. Cuando un proceso invoca un wait, siempre se procede a dormir; en cambio si realiza un **P**, sólo se demora si el contador es 0.

Por otro lado, cuando se invoca un signal, se despierta a un proceso que esté dormido en dicho momento; en cambio **V** incrementa el contador, despertando a un proceso demorado en **P**, o sino al próximo que arribe al **P** (tiene efecto posterior).

6 – Pthreads

- Comenzado el 12 de jul. de 22, de clase n°05, filminas 5 a 23.
- https://drive.google.com/uc?id=1T2TBPMgrc1ax0z_vrDWMiSaSXJOTJ9GR

6.1 – Introducción

Es una API estandarizada que posibilita la manipulación de hilos, el uso explícito de mutex y de variables condición. Sus funciones comienzan con el prefijo “pthread_”.

6.2 – Hilos

- Create: crea un hilo que ejecuta la función pasada por parámetro.
- Exit: se agrega al final de la función pasada para finalizar la ejecución del hilo.
- Join: demora la ejecución hasta que el hilo pasado por parámetro finalice.
- Cancel: solicita la finalización del hilo pasado por parámetro. No-bloqueante.

6.3 – Exclusión Mutua

Se realiza por medio de variables mutex, que tienen 2 estados: locked (bloqueado) y unlocked (desbloqueados). Se inicializan en este último. Pueden ser de 3 tipos: normal (se puede hacer lock solo si está unlocked), recursivo (el hilo que le hizo lock puede volver a hacerlo, incrementando un contador interno del mutex) y error check (devuelve un reporte si un mismo hilo intenta un segundo lock). El **lock** es una operación **atómica.**[1]

La variable mutex debe estar declarada, para luego en el main invocar **mutex_init**, que la inicializa y recibe una serie de atributos (el tipo). Las funciones para bloquear y desbloquear son **mutex_lock** y **mutex_unlock**, que reciben la variable por parámetro.

La función mutex_lock es bloqueante, demorando la ejecución hasta que el estado sea unlocked y se logre hacer un lock. Su contraparte no-bloqueante es **mutex_trylock**, que reduce el overhead (tiempo gastado en coordinación) por espera ociosa.

6.4 – Variables condición

Se usan variables que tienen un predicado booleano. Funciones “pthread_cond”.

[1] **Operación atómica:** *en este caso, un único Thread puede ejecutar la operación (bloqueo) a la vez.*

6.4.1 – Wait

Se pasa por parámetro la variable condición y una variable mutex.

1) Se realiza **unlock** en la variable mutex, otros hilos entrarán a la sección crítica.
2) Se duerme al hilo hasta recibir el signal de la variable condición. ¡No usa CPU!
3) Se realiza **lock** en la variable mutex, para recuperar el acceso a la sección crítica.

Una variante es el "timedwait", que establece un tiempo máximo de espera.

6.4.2 – Signal

Se pasa por parámetro la variable condición. Únicamente despierta al primer hilo que se encuentra dormido para dicha variable. La variante "broadcast" despierta a todos.

6.5 – Simulando Monitores

Pthreads ofrece manejar exclusión mutua mediante mutex, y sincronización por condición mediante variables condición. Los monitores, que tienen procedimientos que se ejecutan en exclusión mutua, se pueden implementar añadiendo una variable mutex por cada monitor (no confundir con cantidad de procedimientos). Luego, cada hilo debe hacer un lock, ejecutar las instrucciones del procedimiento y hacer unlock, siempre.

```
void *escritor (void*)
{  while (true)
     { pthread_mutex_lock (&mutex);
       if (nr>0 OR nw>0)
         { dw = dw +1;
           pthread_cond_wait (& ok_escribir, &mutex);
         }
       else  nw = nw + 1;
       pthread_mutex_unlock (&mutex);
       //Escribe sobre la BD
       pthread_mutex_lock (&mutex);
       if (dw > 0)
         { dw = dw -1;
           pthread_cond_signal(&ok_escribir);
         }
       else
         { nw = nw -1;
           if (dr > 0)
             { nr = dr;
               dr = 0;
               pthread_cond_broadcast(&ok_leer);
             };
         };
       pthread_mutex_unlock (&mutex);
     };
   pthreads_exit();
};
```

```
void *lector(void*)
{  while (true)
     { pthread_mutex_lock (&mutex);
       if (nw>0)
         { dr = dr +1;
           pthread_cond_wait (& ok_leer, &mutex);
         }
       else  nr = nr + 1;
       pthread_mutex_unlock (&mutex);
       //Leer sobre la BD
       pthread_mutex_lock (&mutex);
       nr = nr – 1;
       if (nr == 0 and dw > 0)
         { dw = dw -1;
           pthread_cond_signal(&ok_escribir);
           nw = nw + 1;
         };
       pthread_mutex_unlock (&mutex);
     };
   pthreads_exit();
};
```

7 – Librería OpenMP

- Comenzado el 12 de jul. de 22, de clase n° 05, filminas 36 a 56.
- https://drive.google.com/uc?id=1hbMW8CiW9yYtmdkrPyDPhP74l-jKagQd

7.1 – Introducción

Es una API que reúne las siguientes características:

- Paralelismo basado en Threads (no hay procesos), usando memoria compartida.
- Paralelismo explícito: indico qué partes son paralelas, cómo reparto el trabajo…
- Permite alterar la cantidad de Threads de manera dinámica.
- Es un estándar de programación paralela basada en directivas del compilador, es decir, constructores de alto nivel que admiten una lista de cláusulas.
- Utiliza el Modelo Fork-Join (ver sección 7.3)

7.2 – Directiva Parallel

Comienza con **#pragma omp parallel**, luego tiene una lista de cláusulas (if, las variables locales de cada thread, las compartidas, etc.), y un bloque {…} paralelo.

```
#include <omp.h>
….
/* Suponiendo que se crean 3 threads (además del master) */
#pragma omp parallel reduction(+: sum)
   {  sum = omp_get_thread_num();
   }
/* En este punto sum es igual a 6 */
…
```

Ejemplo: reduction indica que en el JOIN, se realiza la operación especificada con la variable local de cada Thread creado, y el resultado queda guardado en el Thread Master. La función del bloque retorna el número de hilo (0 a 3).

7.3 – Modelo Fork-Join

1) Se comienza con un único proceso, llamado Thread Master.
2) Cuando el Master encuentra un constructor paralelo (directiva parallel), realiza un FORK: crea un grupo de Threads, y él se queda con el id 0 del grupo. [2]
3) Ejecutan en paralelo el bloque encerrado por el constructor de la región paralela.
4) Al finalizar el bloque, se sincronizan y terminan. Sólo continúa el Master.

[2] La cantidad de Threads creados se asigna en OMP_NUM_THREAD, o con *omp_set_num_threads()*

7.4 – Directiva For

Es un constructor de trabajo compartido[3] que divide las iteraciones de un Loop entre los Threads, es decir, permite realizar un paralelismo de **datos**. Admite un conjunto de cláusulas, entre las cuales se destaca **schedule(tipo, N)**, que admite 4 tipos:

- ✓ Static: divide el trabajo en bloques de N elementos, y los reparte a los Threads en forma Round Robin (cíclico). Si N está ausente, habrá un bloque por cada hilo.
- ✓ Dynamic: divide en bloques de N elementos y los reparte bajo demanda, es decir, cada vez que un Thread solicite. Si N está ausente, el bloque es 1 sola iteración.
- ✓ Guided: divide en bloques de tamaño grande, luego se va achicando el tamaño a repartir bajo demanda hasta N elementos o 1 sola iteración si N está ausente.
- ✓ Runtime: tipo determinado por la variable de entorno OMP_SCHEDULE.

7.5 – Directiva Section

Es un constructor de trabajo compartido que permite asignar tareas paralelas que no son iterativas, es decir, secciones de trabajo separadas para paralelismo **funcional**. La declaración de cada sección es mediante **#pragma omp section [cláusulas] {…}**

7.6 – Directiva Single

Es un constructor de trabajo compartido que permite la ejecución de un bloque de código únicamente por un único Thread (el que llega primero). El resto de Threads que lleguen al bloque mientras está en ejecución, deberán esperar a que termine, salvo que se haya indicando la cláusula **nowait** en la directiva: **#pragma omp single nowait {…}**

7.7 – Constructores combinados

Si solo necesito usar una directiva For o Section en un bloque paralelo, puedo usar directamente **#pragma omp parallel for** o **#pragma omp parallel section**.

Además, si la variable de entorno OMP_NESTED está habilitada, se pueden anidar, por ejemplo, directivas Parallel For dentro de otra Parallel For principal; sin embargo esto provoca que se creen Threads por cada iteración del bucle principal.[4]

[3] Significa que no crea nuevos Threads, por lo que debe insertarse dentro de un constructor paralelo.
[4] Hay un doble grado de paralelización, lo que es muy costoso (no recomendable)

8 – Memoria Distribuida

- Comenzado el 14 de jul. de 22, resumido de clases n°06, 07 y 09.

8.1 – Introducción

Una de las formas de la programación concurrente es la programación distribuida, que consiste en la comunicación mediante mensajes. Asume que la ejecución se realiza sobre una arquitectura de memoria distribuida (procesadores con memorias locales), pero también se puede ejecutar sobre una memoria compartida o hibrida.

Entre los procesos sólo se comparten canales, que pueden ser lógicos o físicos.

8.2 – Pasaje de Mensajes

En PMA, los canales se declaran como globales para los procesos. Si cualquiera puede enviar o recibir por alguno de los canales declarados son **mailboxes**. Si solo tiene un único receptor y múltiples emisores son **input-port**. Si tiene un único receptor y un único emisor se denomina **link**. En PMS, todos los canales son de este último tipo.

Tanto en PMA como en PMS los canales son unidireccionales.

8.3 – RPC y Rendezvous

RPC significa “Remote Procedure Call”. El programa se descompone en módulos, que poseen procesos y procedures. Los procedures, a su vez, pueden ser locales (invocados por el mismo módulo) o exportados. Cada módulo puede estar en una máquina distinta, entonces en un llamado intermódulo se crea un nuevo proceso **server** que sirve el llamado, pasando los argumentos como mensajes entre ambos. Sin embargo, los llamados pueden ser concurrentes: debe programarse sincronización dentro del módulo.

Rendezvous combina comunicación y sincronización. Nuevamente, un proceso cliente (llamador) invoca una operación mediante un **call**, pero la misma es servida por un proceso server **ya existente** (no se crea uno nuevo como en RPC). Para ello, el server tiene sentencias de entrada para esperar un llamado y actuar. Además, cada invocación se atiende una por vez (exclusión mutua implícita). El server es un **proceso activo.**

Como similitud, en RPC como Rendezvous los canales son bidireccionales.

9 – Arquitecturas Paralelas

- Comenzado el 14 de jul. de 22, de clase n°09, filminas 4 a 30

9.1 – Introducción

Vamos a ver las diferentes formas de clasificarlas.

9.2 – Espacio de Direcciones

- Memoria compartida: comunicación a través de accesos a memoria compartida, multiprocesadores con sus caché (esquema UMA). Problema de consistencia.
- Memoria distribuida: comunicación mediante intercambio de mensajes, múltiples procesadores conectados por una red, memorias locales. Grado de acoplamiento.

9.3 – Mecanismo de Control

- SISD: se ejecutan instrucciones en secuencia, una por ciclo. La memoria afectada sólo es utilizada por dicha instrucción. La ejecución es determinística.
- SIMD: cada procesador tiene su memoria y ejecutan la misma instrucción pero sobre distintos datos. Un host hace broadcast de la instrucción. Determínistica.
- MISD: cada procesador ejecuta un flujo de instrucciones distinto, pero comparten datos comunes (single data). Ejemplo: múltiples filtros sobre una misma señal.
- MIMD: cada procesador ejecuta un flujo de instrucciones distinto sobre datos que pueden estar en memoria compartida o distribuida. Puede ser SPMD o MPMD.

9.4 – Red de interconexión

- Redes estáticas: con links punto a punto, generalmente para pasaje de mensajes.
- Redes dinámicas: con switches y enlaces, para máquinas de memoria compartida.

9.A – Performance de Memoria

Muchas veces, la limitación se encuentra en memoria y no en el procesador. Por un lado, la **latencia** es el tiempo que transcurre desde el pedido a memoria (request) hasta que los datos están disponibles. Por otro lado, el **ancho de banda** es la velocidad en la que el sistema de memoria puede colocar los datos en el procesador. La idea es explotar la localidad espacial y temporal de los datos para tener mayores aciertos de caché.

10 – Paradigmas de Interacción

- Comenzado el 13 de jul. de 22, de clase n°07, filminas 29 a 41.

10.1 – Master/Worker

Consiste en una “bolsa de tareas independientes”, compartida por un conjunto de workers, cada uno saca una de la bolsa, la ejecuta y puede crear nuevas tareas para poner en la bolsa. En la implementación mediante mensajes, un proceso Manager administra la bolsa: maneja las tareas, se comunica con los workers y detecta el fin de las tareas.

Ejemplo: multiplicar dos matrices ralas (una tarea por cada celda resultado).

10.2 – Algoritmos Heartbeat 

Utiliza un esquema “divide y conquista”, distribuyendo los datos entre workers, y cada uno es responsable de actualizar una parte. Los nuevos valores dependen de los que mantienen los workers o sus vecinos inmediatos. Cada par de vecinos está conectado por canales bidireccionales. Para determinar la topología, se usa una matriz de adyacencia.

Es “Heartbeat” porque periódicamente se expande enviando información, y luego se contrae incorporando información. Se supone que cada paso es un progreso hacia la solución. Sirve para soluciones iterativas a paralelizar. Ejemplo: simular fenómenos.

10.3 – Algoritmos Pipeline

El concepto “pipeline” refiere a un arreglo de procesos workers, donde cada uno actúa como un filtro: recibe datos de un canal de entrada y entrega resultados a un canal de salida. Si los procesos están en procesadores que trabajan en paralelo decimos que es un *esquema a lazo abierto*. Para procesos iterativos (+1 pasada) se usa *esquema circular*; pero si un coordinador realimenta el primero con el último es *esquema cerrado.*

Ejemplo: multiplicar dos matrices por bloques.

10.4 – Prueba/Eco

Análogo concurrente de DFS para árboles y grafos. Un nodo envía un mensaje de forma paralela a todos los sucesores (pruebas) y espera un mensaje de respuesta (eco).

Ejemplo: recorrer redes móviles (no se conoce el número de nodos).

11 – Paradigmas de Programación

- Comenzado el 13 de jul. de 22, de clase n°10, filminas 58 a 74.

11.1 – Introducción

Son clases de algoritmos que comparten una misma estructura de control pero permiten resolver diferentes problemas. Se puede escribir un esqueleto algorítmico.

11.2 – Master/Worker

Consiste en un Maestro que envía mensajes datos de manera iterativa a los workers o esclavos, y recibe resultados de éstos. Según la dependencia de las iteraciones, está el caso en que el maestro necesita los resultados de todos los workers para generar nuevos datos, o bien, que al maestro le llegan datos y genera un nuevo conjunto de datos.

A su vez, la distribución de datos puede ser estática (todos los disponibles según alguna política) o dinámica (bajo demanda). En resumen, siempre un procesador es coordinador, y los otros resuelven problemas asignados. Ejemplo: UI de mediciones.

11.3 – Dividir y Conquistar

Consiste en descomponer el problema general (programa) en subproblemas más chicos, para asignarlos a procesos recursivos que trabajan sobre una parte del conjunto de datos de forma independiente, y luego combinar los resultados. Ejemplo: merge sort.

11.4 – Pipeline

Consiste en descomponer el problema en una serie de pasos. Cada proceso actúa como un filtro: recibe un stream de datos, realiza una tarea sobre el mismo, y entrega resultados a la salida. Si se tienen N pasos, se puede realizar una tarea distinta sobre N conjuntos de datos en simultáneo. Ejemplo: filtrar, etiquetar y analizar imágenes.

11.5 – SPMD

Consiste en realizar un programa único que ejecutará cada nodo (máquina) sobre una parte del dominio de datos. Primero se elige la distribución y después se genera el programa paralelo a partir del secuencial. Ejemplo: multiplicar matrices por fila o celda.

12 – Librería MPI

- Comenzado el 11 de jul. de 22, de clase n°08, filminas 8 a 27.
- https://drive.google.com/uc?id=1tlOM5BaPD2KSlRIUVYWnwO-8SDqLPlE8&export=download

12.1 – Introducción

Es una librería para manejar pasajes de mensajes. Posee más de 125 rutinas, todas con prefijo “MPI_” y retorno de un código MPI_SUCCESS si la operación fue exitosa.

12.2 – Comunicador

Es una variable del tipo MPI_Comm que almacena información sobre qué procesos pertenecen al **dominio** de dicho comunicador. Cada proceso puede pertenecer a uno o más comunicadores. El que incluye a todos es MPI_COMM_WORLD.

12.3 – Rutinas Principales

- MPI_Init: inicializa el entorno MPI, primer llamado a rutinas MPI en procesos.[5]
- MPI_Finalize: cierra el entorno MPI, último llamado a rutinas MPI en procesos.
- MPI_Comm_Size: indica la cantidad de procesos que tiene un comunicador.[6]
- MPI_Comm_Rank: indica el ID del proceso invocador en un comunicador.[7]
- MPI_Send: envía datos a otro proceso de un comunicador. ¡Es bloqueante!
- MPI_Recv: recibe datos de otro proceso de un comunicador. ¡Es bloqueante!

12.4 – Parámetros del Send

- Un puntero a la variable que almacena actualmente el mensaje.
- La cantidad de mensajes a enviar (entero).
- El MPI_Datatype, es decir, cuál es el tipo de dato estandarizado del mensaje.
- El ID del proceso DESTINO, acorde al comunicador utilizado.
- Un tag (entero) que identifique al mensaje para su posterior recepción.
- El comunicador.

[5] Tanto MPI_Init como MPI_Finalize deben invocarse en todos los procesos involucrados.
[6] Se pasa el comunicador como primer parámetro, y devuelve la cantidad por referencia en el segundo.
[7] **ATENCIÓN:** cada proceso puede tener un ID diferente en cada comunicador (entre 0 y size del mismo).

12.5 – Parámetros del Receive

- Un puntero a la variable donde se almacenará el mensaje recibido.
- La cantidad de mensajes a recibir (entero).
- El MPI_Datatype, es decir, cuál es el tipo de dato estandarizado del mensaje.
- El ID del proceso ORIGEN en comunicador. Comodín: MPI_ANY_SOURCE.
- El tag (entero) utilizado para identificar el mensaje. Comodín: MPI_ANY_TAG.
- El comunicador.
- Un puntero a un MPI_Status: contiene el ID origen, el tag y un código de error.

12.6 – Ejemplito Bloqueante

Dos procesos intercambian valores (14 y 25). Solución empleando MPI:

```
# include <mpi.h>
main (INT argc, CHAR *argv [ ] ) {
    INT id, idAux;
    INT longitud=1;
    INT valor, otroValor;
    MPI_status estado;

    MPI_Init (&argc, &argv);

    MPI_Comm_Rank (MPI_COMM_WORLD, &id);
    IF (id == 0) { idAux = 1; valor = 14;}
    ELSE { idAux = 0; valor = 25; }

    MPI_send (&valor, longitud, MPI_INT, idAux, 1, MPI_COMM_WORLD);
    MPI_recv (&otroValor, 1, MPI_INT, idAux, 1, MPI_COMM_WORLD, &estado);
    printf ("process %d received a %d\n", id, otroValor);
    MPI_Finalize ( );
}
```

12.7 – Comunicación No-Bloqueante

Se logra con las rutinas MPI_Isend y MPI_Irecv. Éstas comienzan la operación y devuelven el control de manera inmediata. No garantizan una comunicación exitosa. El único cambio en sus parámetros es que luego del comunicador se indica un puntero a un MPI_Request, que sirve de referencia para luego consultar si la operación finalizó (con la rutina MPI_Test) o si quiero esperar a que termine (MPI_Wait, bloquea el proceso).

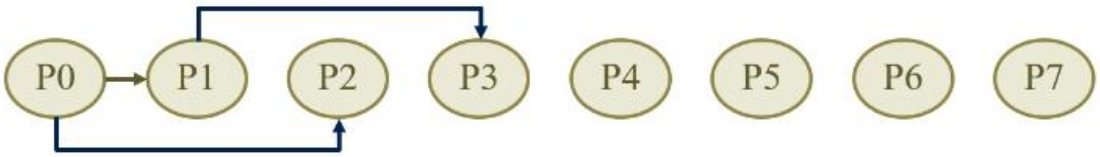
También existe la rutina MPI_Iprobe, que verifica si el mensaje entrante cumple con un origen y tag especificados por parámetro. La MPI_Probe, en cambio, bloquea el proceso hasta que llegue un mensaje que cumpla dichas condiciones.

12.8 – Comunicación Colectiva

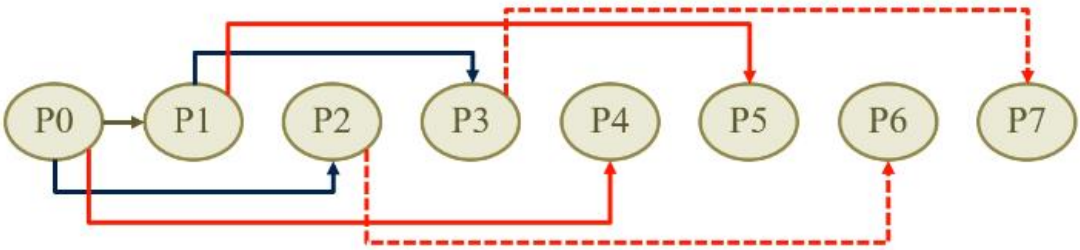
Se refiere a las rutinas que realizan operaciones sobre un **grupo de procesos** que pertenecen a un comunicador. Todos los procesos involucrados deben invocar la misma rutina. Aumentan la eficiencia en comparación a la comunicación punto a punto:

- Supongamos que existen 8 procesos, y se quiere realizar un Broadcast.
- Iteración 1: el primer proceso (P0) envía el mensaje al proceso P1.

- Iteración 2: los procesos P0 y P1 envían el mensaje a P2 y P3, respectivamente.



- Iteración 3: P0, P1, P2 y P3 envían a P4, P5, P6 y P7, respectivamente.



- Se observa entonces que el tiempo de ejecución tiene un **orden logarítmico.**
- En contraparte, si únicamente P0 enviara mensajes al resto, sería orden lineal.

12.9 – Sincronización por Barrera

Es la operación colectiva MPI_Barrier. Recibe el comunicador por parámetro, y su función es bloquear el proceso invocador hasta que todos lleguen a dicho punto.

12.10 – Broadcast

Es la operación colectiva MPI_Bcast. Envía el mensaje a todos los procesos en el comunicador, incluso el que lo genera. El parámetro “origen” debe coincidir en todos.

13 – Métricas de Rendimiento

- Comenzado el 13 de jul. de 22, de clase n°10, filminas 51 a 56

13.1 – Introducción

A falta de un modelo unificador, el tiempo de ejecución depende del tamaño de la entrada (datos), de la arquitectura y número de procesadores.

13.2 – Speedup (S)

Es el cociente entre el tiempo de ejecución del algoritmo serial más rápido (Ts) y el tiempo de ejecución paralelo del algoritmo a medir (Tp). Es una métrica independiente del tamaño del problema, tiene que ser el mismo para ambas medidas de tiempo.

El speedup óptimo depende de la arquitectura, y se calcula como la sumatoria de la potencia de cálculo de cada procesador “i”, dividida la potencia de cálculo del mejor.

El speedup puede ser líneal (perfecto), sublineal (menor al óptimo) o superlineal (mayor al óptimo). Este último se puede dar por 3 razones: una mala elección del mejor algoritmo secuencial, búsqueda en árboles, o por gran uso de la memoria caché.

13.3 – Eficiencia (E)

Es el cociente entre el speedup y el speedup óptimo. Toma valores entre 0 y 1, y mide la fracción de tiempo en que los procesadores son útiles para el cómputo. En otras palabras, mide el aprovechamiento de la arquitectura.

13.4 – Limitaciones

En primer lugar, la Ley de Amdahl enuncia que siempre existe un porcentaje de código secuencial en los programas paralelos. Puede haber desbalances de carga si los procesadores son heterógeneos, produciendo esperas ociosas. Puede haber overhead, es decir, ciclos adicionales de CPU utilizados para creación de procesos y sincronización.

Un mal diseño lleva a algoritmos inadecuados, gran contención de memoria (debe rediseñarse el código para una mayor localidad de datos). También puede ocurrir que el algoritmo no sea igual de eficiente para distintos tamaños del problema.