Sistemas Paralelos – Resumen

Clase 1 – Introducción a los sistemas paralelos

Procesamiento Secuencial

- Software desarrollado para ejecución secuencial.
- Problema se divide en instrucción que se ejecutan secuencialmente una después de la otra en un solo procesador.

Procesamiento Paralelo

- Múltiples unidades de procesamiento para resolver un problema computacional se divide en partes que pueden ser concurrentes.
- Cada parte se dividen en instrucciones que se pueden ejecutar simultáneamente en diferentes procesadores.
- Se necesita mecanismo de control/coordinación.
- Permite:
    - Resolver problemas más grandes o complejos.
    - Proveer concurrencia.
    - Ahorrar tiempo y/o dinero
    - Hacer mejor uso de los recursos de hardware
- Problemas que lo usan:
    - Ciencia e Ingeniería.
    - Industria y comercio.

Procesamiento Concurrente, Paralelo y Distribuido

- Algunos autores hacen las siguientes distinciones:
    - Programa concurrente:
        - Múltiples tareas puede estar avanzando en cualquier instante de tiempo.
    - Programa paralelo
        - Múltiples tareas se ejecutan simultáneamente cooperando para resolver un problema.
        - Busca reducir el tiempo de ejecución usando varios procesadores al mismo tiempo.
            - Sigue siendo fundamental desarrollar un software que aproveche el hardware subyacente.
            - Una consecuencia de ello es la gran dependencia entre el software y el hardware.
            - El hardware evoluciona.
    - Programa distribuido:
        - Múltiples tareas que se ejecutan físicamente en diferentes lugares cooperan para resolver uno o más problemas.
        - Conjunto de computadoras independientes interconectadas que cooperan compartiendo recursos.

        - En un principio no se fabrican como tales, lo que da lugar a la heterogeneidad, lo que llega a ser un problema.
        - Problemas: no hay reloj único, requiere planificación, escalabilidad depende de las comunicaciones.
- Paralelo y distribuido son programas concurrentes.
    - Características comunes:
        - Usan múltiples procesadores.
        - Procesadores interconectados por una red.
        - Múltiples tareas evolucionan al mismo tiempo y cooperan/compiten.
    - Diferencias:
        - Programa paralelos se descomponen en tareas que se ejecutan
        - al mismo tiempo en el mismo lugar mientras que los distribuidos en distintos lugares.
- Concurrencia es concepto de software y especificar la concurrencia implica especificar los procesos concurrentes, su comunicación y sincronización.

Computo de alto rendimiento

- También llamado Cómputo de Alto Desempeño o HPC por High-Performance Computing
- Uso de sistemas de extraordinario poder computacional y de técnicas de procesamiento paralelo para la resolución de problemas complejos con alta demanda computacional
- Involucra una amplia gama de temas: algoritmos, técnicas de programación, aplicaciones, hardware, redes, herramientas, etc.
- Se diferencia con procesamiento paralelo.
- Se lo usa en:
    - Bioingeniería
    - Química
    - Genómica
    - Aeronáutica
    - Computación gráfica
    - Modelización financiera
    - Analítica de datos
    - Medicina

Clasificación de las plataformas (arquitectura subyacente) de cómputo paralelo

- Se clasifican por:
    - Mecanismo de control
        - SISD:
            - Instrucciones son ejecutadas en forma secuencial, una por ciclo de reloj.
            - Siempre los datos afectados son a los que hace referencia la instrucción.
            - La ejecución se vuelve determinística.

        - Tipo más antiguo de computadoras:
            - Mainframes, monoprocesadores.
    - SIMD:
        - Unidades de procesamiento ejecutan la misma instrucción sobre diferentes datos.
        - Ejecución sincrónica y determinística.
        - Hardware simplificado (comparten control).
        - Pueden deshabilitarse algunas unidades para que ejecuten o no instrucciones.
        - Adecuado para problemas con alto grado de regularidad.
        - GPUs.
    - MISD:
        - Unidades de procesamiento ejecutan diferentes instrucciones sobre el mismo dato.
        - No existen maquinas reales.
    - MIMD:
        - Unidades de procesamiento ejecutan diferentes instrucciones sobre diferentes datos.
        - Ejecución puede ser sincrónica o asincrónica, determinística o no determinística.
        - Pueden ser máquinas de memoria compartida o de memoria distribuida.
        - Clase más común de máquina paralela.
        - procesadores multicore, clusters, multiprocesadores, grids, supercomputadoras
- Por la organización física: de acuerdo al espacio de direcciones que tiene cada procesador. Tiene en cuenta la visión de la memoria principal de cada procesador.
    - Memoria compartida:
        - Procesadores pueden acceder a toda la memoria como un único espacio de direcciones global.
        - Procesadores trabajan independientemente pero comparten memoria.
        -
        - Se necesita mecanismo coherencia de cache.
        - Subclasificación en:
            - Acceso uniforme a memoria (UMA).
            - Acceso no uniforme a memoria (NUMA).
        - Ventajas:
            - Comunicación entre procesadores rápida y uniforme.
            - Programación suele ser más fácil.
        - Desventaja:
            - Se debe asegurar el correcto acceso a los datos.
            - Falta de escalabilidad entre procesadores y memoria.
    - Memoria distribuida:

- Procesadores trabajan independientemente cada uno con su memoria.
- El programador es el responsable de definir cómo y cuándo serán comunicado los datos entre procesos.
- No se requiere un mecanismo de coherencia de caché.
- Se requiere una red de comunicación (red de interconexión) para poder conectar a los procesadores.
    - Puede ser Ethernet u otras.
- Ventajas:
    - Memoria escala con el número de procesadores.
    - Como no hay protocolo coherencia de caché cada procesador accede más rápido a los datos.
    - Buena relación costo-rendimiento.
- Desventajas:
    - Acceso NUMA: se tarda más en acceder a datos en nodos remotos
    - Manejo de comunicaciones explicito.
    - Espacio de direccionamiento distribuido puede condicionar la programación.

- Memoria híbrida
    - Los sistemas más grandes y rápidos del mundo de la actualidad combinan características de ambos modelos.
    - Múltiples máquinas de memoria compartida son interconectadas entre sí para permitir que sus procesadores puedan comunicarse.
    - Ventajas y desventajas:
        - Las mismas de ambos modelos.
        - Soluciona el problema de escalabilidad de memoria compartida.
        - Aumenta la complejidad de programación.

Modelos de programación paralela

- Memoria compartida.
    - Todas las tareas concurrentes acceden a una memoria compartida que permite realizar la comunicación y sincronización.
    - El programador en general no maneja la distribución de los datos ni lo relacionado a la comunicación de estos.
    - Ventaja:
        - Transparencia para el programador.
    - Desventaja
        - A veces es necesario trabajar sobre esos aspectos para mejorar el rendimiento.
        - Difícil predecir performance.
- Pasaje de mensajes.
    - Cada procesador tiene su propio espacio de direcciones.
        - Espacio de direcciones particionado.
    - Toda interacción requiere la cooperación de dos procesos.

    - Intercambio de mensajes para:
        - Intercambio explícito de datos.
        - Sincronización.
    - Ventajas:
        - El programador tiene total control
        - Puede implementarse eficientemente en muchas arquitecturas paralelas.
        - Más fácil de predecir el rendimiento (y por lo tanto optimizar).
    - Desventajas:
        - Mayor complejidad de algoritmos.
- Los modelos pueden usarse en cualquier arquitectura, sin embargo, el uso de un modelo que no resulte "natural" para la arquitectura subyacente puede llevar a mal rendimiento.

Evolución del poder computacional

- El aumento de transistores y el aumento de la frecuencia del reloj lleva a mejoras en los procesadores.
    - Aumento de transistores permite implementar paralelismo a nivel de instrucciones (ILP).
        - Pipelining: solapa las diferentes etapas de la ejecución de instrucciones, reduciendo el tiempo de ejecución total.
            - La velocidad de un pipeline está limitada por la duración de su etapa más costosa.
            - No es útil tener muchas etapas. Se suelen usar entre 14 y 15, mas no es muy eficiente debido a que estadísticamente hay un salto condicional cada 5-6 instrucciones.
            - Para mejorar la tasa de ejecución de instrucciones se pueden usar varios pipelines (ejemplo procesador con 2 pipelines emite simultáneamente 2 instrucciones, esto se llama superescalar).
            - El rendimiento de un procesador superescalar está limitado por la cantidad disponible de paralelismo a nivel de instrucciones.
                - Al momento de realizar la planificación de instrucciones, se deben tener en cuenta:
                    - Dependencia verdadera de datos.
                    - Dependencia de recurso.
                    - Dependencia de salto.
                - El planificador analiza el conjunto de instrucciones y emite aquellas que pueden concurrentemente:
                - Emisión en orden: instrucciones son ejecutadas en el orden en que aparecen en la cola. Simple pero limita la emisión.
                - Emisión fuera de orden: el procesador reordena las instrucciones en la cola. Se puede alcanzar el máximo rendimiento posible. Es el que se usa en la actualidad. Complejo.

        - Instrucciones SIMD: incorpora unidades de procesamiento vectorial que ejecutar una instrucción sobre diferentes datos en un ciclo de reloj.
    - Aumento de frecuencia de reloj permite tener más ciclos por segundo, lo que posibilita ejecutar más instrucciones por segundo.
- Limite en los 2000s debido a 3 factores:
    - Memory Wall:
        - La memoria es más lenta que los procesadores, por lo que el rendimiento está dominado por la velocidad de memoria.
    - ILP Wall:
        - Es posible agregar más unidades funcionales en chip, pero no se mejora el rendimiento porque no se puede extraer más ILP de los programas.
    - Power Wall:
        - Potencia de los procesadores se transforma en calor que debe disiparse. No es posible incrementar la frecuencia de reloj con los mecanismos de refrigeración actuales.
- Necesario buscar otras alternativa de diseño de los para poder incrementar el rendimiento. Se opto por integrar dos o más núcleos (también llamados cores) más simples en un sólo chip (multicores).
    - Mejoran el rendimiento sin necesidad de aumentar la frecuencia de reloj, lo que los vuelve más eficientes energéticamente.
    - Los primeros procesadores eran dos procesadores mononúcleo en el mismo chip.
    - Siguiente generaciones incrementaron el número de núcleos e incorporado niveles de caché L2 y L3.
- Con respecto al software debido a eso se tuvieron que reprogramar aplicaciones, sistemas operativos, librerías, middlewares, etc.
- Una técnica complementaria al multicore es el multihilado (Simultaneous Multi-Threading):
    - Un único hilo de ejecución no resulta suficiente para aprovechar potencia de los procesadores superescalares.
    - La solución es tener más de un hilo de ejecución al mismo tiempo en el procesador.
        - No pueden usar la misma unidad funcional al mismo tiempo (Integer)
    - Se puede tener multicore con/sin SMT o monocore con/sin SMT.
        - El número de hilos hardware suele ser 2 o 4.
    - Mejora productividad cuando se pueden intercalar instrucciones de múltiples hilos.
    - Si los hilos usan los mismos recursos se pierde rendimiento.
    - Entonces la ganancia depende del programa.

Jerarquía de memoria en procesadores multicore

- Los procesadores multicore suelen tener memoria caches de múltiples niveles
    - El nivel 1 siempre es privado.
    - Los siguientes dependen de la arquitectura.

- Ventajas de las cachés privadas : el acceso a los datos es más rápido y se reduce la competencia en el acceso a los recursos.
- Ventajas de las cachés compartidas: hilos en diferentes núcleos pueden compartir datos que están en la misma caché y hay más espacio de caché disponible si se ejecutan pocos hilos en el procesador.

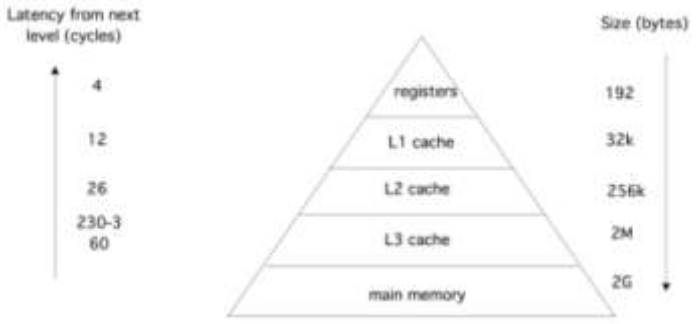
Clusters

- Colección de computadoras individuales interconectadas vía algún tipo de red, que trabajan en conjunto como un único recurso integrado de cómputo.
- Cada nodo es un sistema de computo con hardware y SO propio.
- Puede ser homogéneo o heterogéneo y la red es una Ethernet u otra.
- Buena relación costo-rendimiento y sencillos.
- En la actualidad, la mayoría de los grandes sistemas de cómputo se basan en clusters de nodos multi/many-core.
- Como son arquitecturas distribuidas usan el pasaje de mensajes.
    - La incorporación de procesadores multicore a los clusters lleva a la combinación de pasaje de mensajes con memoria compartida.

Clase 2 – Sistema de memoria

Memory Wall

- Los dos parámetros fundamentales del sistema de memoria a tener en cuenta son:
    - Latencia: tiempo que transcurre desde que se solicita el dato hasta que el mismo está disponible.
        - Se reduce usando caches:
            - Memorias de alta velocidad y baja capacidad que (usualmente) están integradas al chip.
            - Actúan como memoria intermedia entre los registros de la CPU y la memoria principal.



            - Entonces de disminuye la latencia maximizando el número de datos que se acceden desde de la caché.
            - Cuando la CPU necesita un dato, primero revisa si está en la cache:
                - Si esta, se produce un cache hit y el pedido se satisface rápidamente, con baja latencia.

        - Si no está, se produce un cache miss y debe ser buscado en memoria RAM. Luego se copia en la caché antes pasar a los registros.
    - La tasa de hits incide directamente en la latencia global del sistema.
- La mejora obtenida por la inclusión de la caché se basa en la suposición de que habrá una repetición de referencias a determinados datos en una ventana de tiempo pequeña (localidad temporal)
- Ancho de banda: velocidad con la que los datos pueden ser transferidos desde la memoria al procesador.
    - Determinado por el ancho de banda del bus de memoria y las unidades de memoria.
    - Para mejorarlo se incrementa el tamaño de los bloques de memoria que se transfieren por ciclo de reloj.
    - La mejora de rendimiento es posible dado que las sucesivas instrucciones usan datos que se encuentran consecutivos en la memoria (localidad espacial)

- El sistema de memoria requiere l unidades de tiempo (latencia) para obtener b unidades de datos (b es el tamaño del bloque medido en bits, bytes o words)
- Explotar localidad espacial y temporal de datos permite amortizar la latencia e incrementar el ancho de banda efectivo.
- La relación entre el número de instrucciones y el número de accesos a memoria es un buen indicador temprano del rendimiento efectivo del sistema.
- La organización de los datos en la memoria y la forma en que se estructura el código pueden impactar significativamente en el rendimiento final del sistema

Arreglos multidimensionales y su organización en memoria

- Existen 2 maneras en que los datos de un arreglo son almacenados en memoria:
    - Por filas.
    - Por columnas.
- Interesa que:
    1. No imponga un tamaño máximo por la forma en que está declarado el arreglo.
    2. Se pueda elegir cómo se organizan los elementos en memoria.
    3. Sus datos estén contiguos en la memoria.
    4. Si permite cambiar su tamaño en ejecución, mejor.
- Con arreglos en C se consigue:
    - Arreglo estático: punto 4.

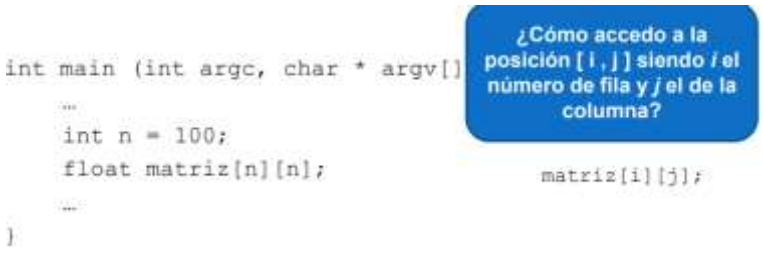
```
#define N 100

int main (int argc, char * argv[]
    …
    float matriz[N][N];
    …
}
```

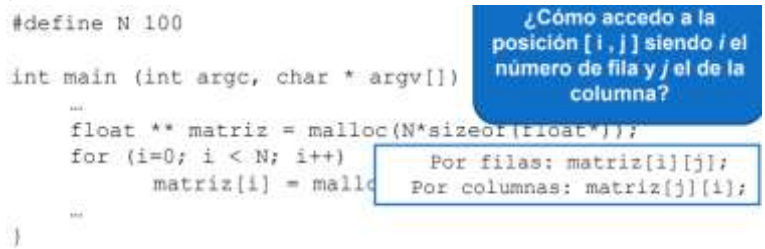
¿Cómo accedo a la posición [i , j] siendo *i* el número de fila y *j* el de la columna?

```
matriz[i][j];
```

    - Arreglo de longitud variable: punto 4.



- Arreglo dinámico como vector de punteros a filas/columnas: todos menos punto 4.

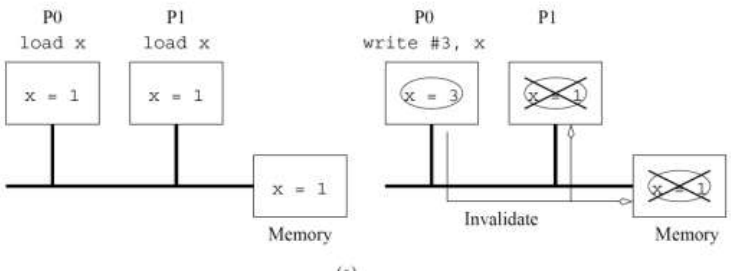


- Arreglo dinámico como vector de elementos: todos.

- Arreglo dinámico como vector de elementos: al almacenar una matriz como un vector de elementos, el cálculo del índice para acceder a sus elementos depende de cómo estén organizados.
  - Cuando una matriz está organizada por filas, se debe multiplicar el número de fila por la cantidad de elementos de cada fila (cantidad de columnas) y sumarle el número de columna.
    - matriz [i*N+j]
  - Cuando una matriz está organizada por columnas, se debe multiplicar el número de columna por la cantidad de elementos de cada columna (cantidad de filas) y sumarle el número de fila.
    - matriz [j*N+i]
  - Ventajas:
    - Favorece al aprovechamiento de la localidad de datos.
    - Hace posible el uso de instrucciones SIMD.
    - Facilita el intercambio de arreglos entre programas escritos en diferentes lenguajes.

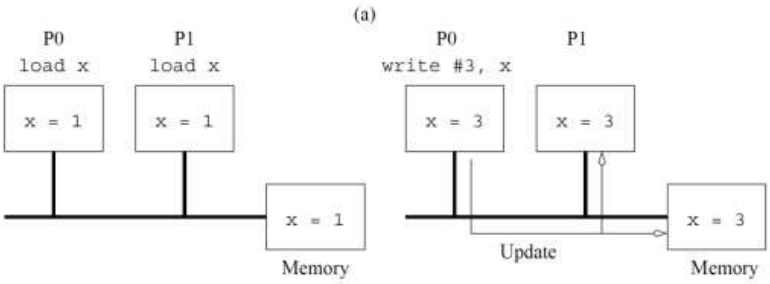
Coherencia de caché en arquitecturas multiprocesador

- Se requiere hardware específico para mantener consistencia de las múltiples copias de un dato en las maquinas de memoria compartida.
- Este debe asegurar que todas las operaciones realizadas son serializables.
- Dos protocolos:
  - Invalidación:
    - Si se tienen dos copias y se cambia una, se invalida la que no es válida.



- Conviene usarse cuando un procesador lee un dato una vez y no vuelve a usarlo.
- Pueden producir ocio ante la espera de actualización de un dato.
- Hoy en día la mayoría de los protocolos son de invalidación.

- Actualización:
  - Si se tienen dos copias y se cambia una, se actualiza la que no es valida y la de memoria.



- Conviene cuando dos procesadores trabajan sobre la misma variable en forma alternada.
- Pueden producir overhead por comunicaciones innecesarias.

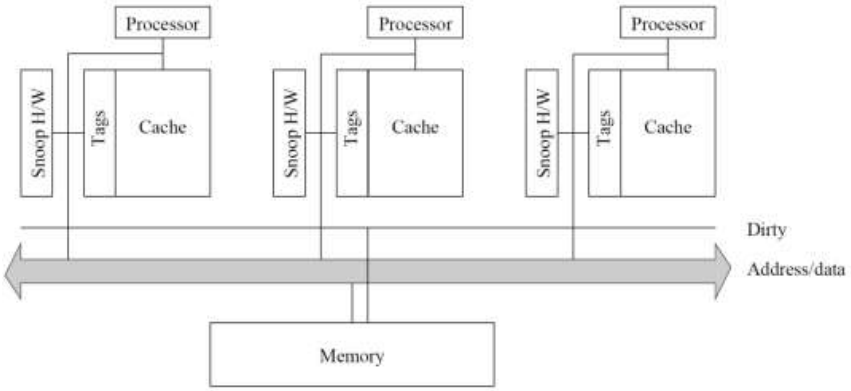
Protocolos de coherencia de caché basados en invalidación

- Cada copia se asocia con uno de 3 estados:
  - Compartida (shared):
    - Hay múltiples copias válidas del dato (en diferentes memorias).
    - Ante una escritura, pasa a estado sucia donde se produjo mientras que el resto se marca como inválida.
  - Inválida (invalid)
    - La copia no es válida.
    - Ante una lectura, se actualiza a partir de la copia válida (la que está en estado sucia)
  - Sucia (dirty)
    - La copia es válida y se trabaja con esta.

Implementación de protocolos de coherencia de caché

- Existe una variedad de mecanismos de hardware para implementar protocolos de coherencia de caché:
  - Sistemas snoopy
    - Asociado a sistemas multiprocesador interconectados con alguna red broadcast
      - La caché de cada procesador mantiene un conjunto de tags asociados a sus bloques que su estado.

- Todos los procesadores monitorizan el bus lo que permite realizar las transiciones de estado de sus bloques:
    - Cuando el hw snoopy detecta una lectura sobre un bloque de caché marcado como sucio, entonces toma el control del bus y cumple el pedido.
    - Cuando el hw snoopy detecta una escritura sobre un bloque de datos del cual tiene copia, entonces la marca como inválida.



- Si cada procesador opera sobre datos disjuntos, entonces los mismos pueden ser cacheados, ya que no agrega overhead.
- Si diferentes procesadores realizan lecturas y escrituras sobre el mismo dato, se genera tráfico en el bus para poder mantener la coherencia de los datos que le llega a todos los procesadores, porque la red es broadcast (esto genera un cuello de botella)
    - Una solución es sólo propagar las operaciones a los procesadores involucrados, lo cual requiere mantener un registro de qué datos tiene cada procesador → Sistemas basados en directorios

- Sistemas basados en directorios
    - La memoria principal incorpora un directorio que mantiene información de estado sobre los bloques de caché y los procesadores donde están cacheados.
    - Permite que sólo aquellos procesadores que tienen un determinado dato queden involucrados en las operaciones de coherencia.
    - Si los procesadores operan sobre datos disjuntos, las peticiones pueden cumplirse localmente.
    - Cuando múltiples procesadores trabajan sobre los mismos datos, se generan operaciones de coherencia que provocan overhead adicional para mantener el directorio actualizado.
        - Como el directorio está en memoria se genera overhead por la competencia en el acceso al recurso.
        - La cantidad de memoria requerida por el directorio podría convertirse en un cuello de botella a medida que se tienen más procesadores.

                - Una solución posible es particionar el directorio → Sistemas basados en directorios distribuidos
    - Sistemas basados en directorios distribuidos:
        - En arquitecturas escalables, donde la memoria esta físicamente distribuida.
        - Cada procesador es responsable de mantener la coherencia de sus propios bloques.
            - Cuando un procesador desea leer un bloque por primera vez, debe pedírselo al dueño, quien redirige el pedido de acuerdo a la información del directorio.
            - Cuando un procesador escribe un bloque de memoria, envía una invalidación al dueño, quien luego la propaga a todos aquellos que tienen una copia.
        - Se alivia la competencia en el acceso teniendo un sistema más escalable.
        - La latencia y el ancho de banda de la red de interconexión se convierten en los cuellos de botella.

Costos de comunicación

- Uno de los mayores overheads proviene de la comunicación.
- Depende de múltiples factores y no sólo del medio físico: modelo de programación, topología de red, manejo y ruteo de datos, protocolos de software asociados.
- Costos diferentes según la forma de comunicación:
    - Modelo simplificado para pasaje de mensajes: tcomm = ts + m.tw
        - ts es el tiempo requerido para preparar el mensaje.
        - m es el tamaño del mensaje medido en palabras (words).
        - tw es el tiempo requerido para transmitir una palabra.
    - Memoria compartida: difícil modelar costos por múltiples factores.

Clase 3 – Programación en memoria compartida – Pthreads

- Como se mencionó anteriormente en las plataformas de memoria compartida:
    - Los procesadores se comunican leyendo/escribiendo variables en la memoria compartida.
    - Los módulos de memoria pueden ser locales o globales.
    - Subclasificación UMA o NUMA.
    - Necesidad de mecanismo de coherencia de cache.
    - Se usa modelo de memoria compartida
        - El problema es que la sincronización está a cargo del programador.
            - Esta sincronización disminuye eficiencia.
        - Es importante la localidad de los datos para el rendimiento.
        - En algunos lenguajes se tiene que actuar sobre la localidad de los datos, en otros reestructurar el código.
        - Puede usarse pasaje de mensajes.
- Modelos de programación proveen soporte para expresar la concurrencia y sincronización:
    - Los basados en procesos suponen datos locales de cada proceso.

- Los basados en threads suponen que toda la memoria es global → Pthreads. Primitivas de bajo nivel.
- Los basados en directivas extienden el modelo basado en threads para facilitar su manejo (creación, sincronización, etc.) → OpenMP. Constructores de alto nivel.

Modelo de hilos

- Un thread es un único hilo de control en el flujo de un programa.
- Tienen acceso a la memoria compartida.
- Tienen su propia memoria privada.
- Ventajas del modelo de hilos frente al de procesos
  - "Liviandad" → Rendimiento: los hilos son más livianos que los procesos permitiendo una intercomunicación y cambio de contexto más rápido.
  - Ocultamiento de latencia → Multi-tasking: múltiples hilos en ejecución reducen la latencia.
  - Planificación y balance de carga: se pueden tener varios hilos que minimizan el overhead por ociosidad y facilitan la distribución de trabajo.
  - Facilidad de programación y uso extendido.
  - Portabilidad.

POSIX Threads

- Hasta los 90s existían varias APIs incompatibles entre ellas para el manejo de hilos.
- Surge POSIX Threads que es un conjunto de tipos de datos y funciones para el lenguaje de programación C y se convierte en la API estándar.
- Las rutinas mas usadas se dividen en 3 grupos:
  - Manejo de threads.
  - Mutexes: mecanismos para exclusión mutua.
  - Variables condición: mecanismos para sincronización por condición.

Manejo de Threads

Creación Hilos

- Inicialmente hay un solo hilo main. El resto deben ser creados por el programador con pthread_create que crea un hilo y lo pone en ejecución.
- Los hilos son pares y pueden crear otros hilos.
- No hay jerarquías o dependencias predefinidas entre los hilos.
- Se pasan los argumentos a través de la función de creación, se puede usar un struct o un vector para pasar varios.

Terminación de hilos

- Con la función pthread_exit que finaliza la ejecución del hilo y retorna un valor.

Join de hilos

- El hilo que invoca la función de creación continúa con su ejecución luego del llamado. Para evitar que el programa termine de forma incorrecta se usa

pthread_join que bloquea al llamador hasta que el hilo pasado como argumento termine.

Mutexes

- La comunicación es implícita, hay que programar la sincronización.
- Las secciones criticas se implementan usando mutex_locks que tienen dos estados: locked (bloqueado) y unlocked (desbloqueado).
    - Se usa pthread_mutex_lock para bloquear. Solo un hilo puede bloquear un mutex_lock (lock es una operación atómica).
    - Todos los lock deben iniciar como desbloqueados.
    - pthread_mutex_unlock para desbloquear.
    - pthread_mutex_init para iniciar el lock.
- Tres tipos de locks que pueden setearse:
    - Normal: no permite que un hilo que lo tienen bloqueado vuelva a hacer un lock sobre él (deadlock).
    - Recursive: si permite que un hilo que lo tienen bloqueado vuelva a hacer un lock sobre él (simplemente incrementa una cuenta de control).
    - Error Check: responde con un reporte de error al intento de un segundo bloqueo por el mismo hilo.
- Tener cuidado con los locks ya que pueden degradar el rendimiento debido a que representan puntos de serialización.
    - Con trylock se puede reducir la espera ociosa ya que retorna el control informando si pudo hacer o no el lock.

Variables condición

- Uso indiscriminado de locks puede provocar un overhead.
- La solución es usar variables de condición que permiten que uno o más hilos se autobloqueen hasta que se alcance un estado determinado del programa.
- Estas variables están asociadas a un predicado (estado) que cuando se convierte en verdadero avisa a los hilos sobre el cambio de estado de la condición. Si el predicado es falso, el hilo espera en la variable condición bloqueándose.
- Una única variable condición puede asociarse a varios predicados (dificulta comprensión)
- Variable condición siempre tiene asociado un lock.
- pthread_cond_wait (condición, mutex) bloquea al hilo hasta que sea despertado. Para invocar el hilo debe tener el control del mutex asociado. Una vez dormido en la variable condición, el mutex se libera. Cuando el hilo recibe una señal espera a que el mutex esté disponible para continuar su ejecución.
- pthread_cond_signal despierta al hilo dormido en la condicion. Para invocar el hilo debe tener el control del mutex asociado. El mutex asociado se libera.
- pthread_cond_init
- pthread_cond_destroy
- Variantes wait y signal:
    - pthread_cond_timedwait duerme por un determinado tiempo.
    - pthread_cond_broadcast despierta a todos los hilos.

Barreras

- Para implementar puntos de sincronización que involucren a múltiples hilos:
    - pthread_barrier_wait en donde el llamador se bloquea hasta que la cantidad de hilos especificada en la barrera haya alcanzado ese punto. La cantidad se especifica en la inicialización.

Semáforos

- Estructura de datos que permite sincronizar hilos.
- POSIX definió una API semaphore.h
- sem_t para declarar un semáforo; sem_init para inicializar; sem_wait para decrementar (P); sem_post para incrementar (V); sem_destroy para destruir.

Planificación de hilos

- Responsabilidad del SO pero el programador puede influenciar usando atributos de planificación.
- La prioridad de planificación de un hilo determina qué nivel de privilegio tendrá.
- El planificador mantiene una cola por cada prioridad. Cuando se tiene que elegir se hace de la cola de mayor prioridad. Si hay varios hilos se elige uno de ellos de acuerdo a la política de planificación.
- Para asignar y recuperar los atributos de planificación:
    - pthread_attr_setschedparam y pthread_attr_getschedparam.
- Para asignar y recuperar la prioridad mínima y máxima de una de terminada política de planificación
    - sched_get_priority_min y sched_get_priority_max.
- La política de planificación determina cómo se ejecutan y comparten recursos los hilos de una misma prioridad:
    - SCHED_FIFO: el hilo se ejecuta hasta que termina, se bloquea o hasta que un hilo de mayor prioridad pueda ejecutarse. Los de misma prioridad son ejecutados en orden.
    - SCHED_RR: similar a SCHED_FIFO pero los hilos se ejecutan a lo sumo una determinada cantidad de tiempo.
    - SCHED_OTHER: política adicional, no definida en el estándar.

Clase 4 – Programación en memoria compartida – OpenMP

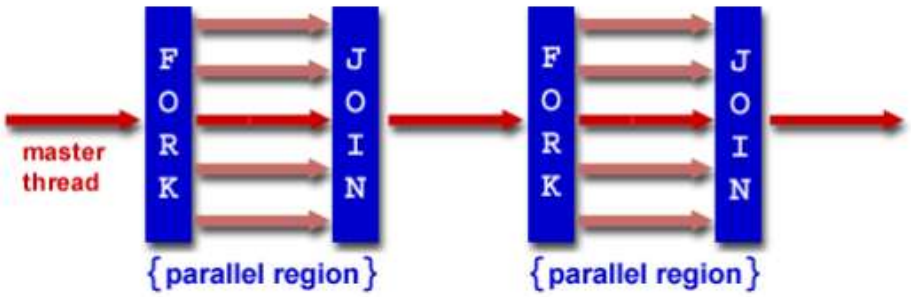
- Modelos basados en directivas la idea básica es liberar al programador del manejo explícito de hilos.

OpenMP

- Tiene 3 componentes primarios: directivas, funciones de librerías y variables de entorno.
- Las directivas (que son traducidas a código Pthreads) proveen soporte para concurrencia, sincronización y manejo de datos obviando el uso explícito de locks, variables condición, alcance de los datos e inicialización de threads.
- Fue creado con el objetivo de brindar un mayor nivel de abstracción.
- Sigue una filosofía incremental de desarrollo.
- Sintaxis de las directivas: #pragma omp nombre_directiva [lista de cláusulas]

Modelo Fork-Join

- Usa el modelo Fork-Join:
    - Se comienza con un único hilo (hilo master).
    - Fork: al encontrar un constructor paralelo el hilo master crea un grupo de hilos.
        - El bloque encerrada por el constructor de la región paralela es ejecutado por todos los hilos.
    - Join: cuando el conjunto de hilos finaliza el bloque paralelo, se sincronizan y terminan, continuando únicamente el hilo master.



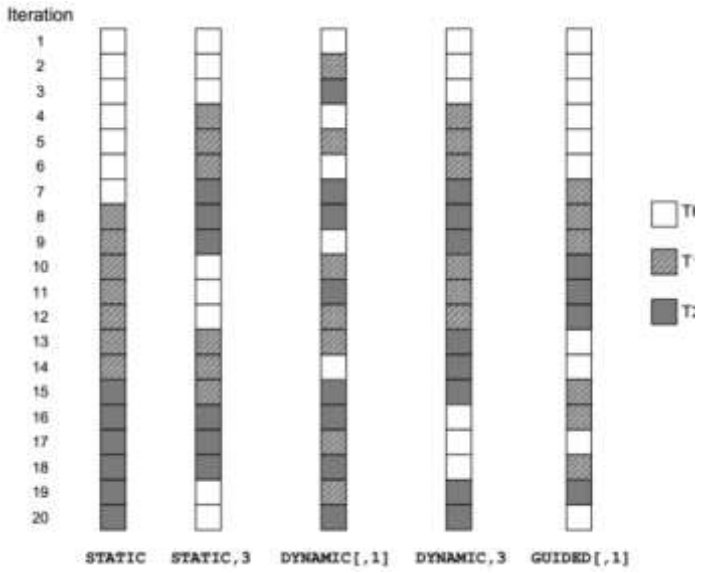
Constructor parallel

- #pragma omp parallel [lista de cláusulas]
    - Admite cláusulas que determinan cuáles datos serán privados a cada hilo y cuáles serán compartidos.
- Es el constructor más importante.
- Permite especificar la región paralela.
- Asegura la creación aunque la distribución de trabajo es responsabilidad del programador.
- Cada hilo mantiene un ID único (el 0 siempre es el del hilo master).
- Al final de la región paralela hay una barrera implícita. Sólo el hilo master continúa con la ejecución.
- Cláusulas private y firstprivate:
    - Las variables privadas de un hilo se especifican mediante la cláusula private:
        - #pragma omp parallel private(lista_de_variables).
        - Crea una copia local a cada hilo de cada variable.
            - Sólo puede ser accedida y modificada por el hilo que la posee.
    - Variante: firstprivate.
- Cláusulas shared y default:
    - Las variables compartidas se especifican en la cláusula shared:
        - #pragma omp parallel shared(lista_de_variables)
        - Todos los hilos podrán leer y modificar la variable original.
        - Por defecto, todas las variables son compartidas.
            - #pragma omp parallel default(shared|private|none) para alterar esto.

        - Si es none el programador debe decir si es compartida o privada.
- Cláusula num_threads:
    - Para especificar el número de hilos a crear
    - Si no se especifica se usa la variable de entorno OMP_NUM_THREADS.
- Cláusula if:
    - Permite generan los hilos en base a la evaluación de una expresión escalar. Se paraleliza solo cuando vale la pena. Si la expresión es falsa, se ejecuta en forma secuencial.
- Parallel puede ser usada en conjunto con otras directivas para trabajo compartido (for y sections).
    - Se pueden combinar con la directiva parallel.

Constructor for

- Divide las iteraciones de un bucle entre los hilos (paralelismo de datos).
- #pragma omp for [lista de cláusulas] for (init_exp; check_exp; mod_exp)
    - Las iteraciones deben ser independientes entre sí.
    - El número de iteraciones debe ser conocido de antemano.
    - La variable índice se vuelve privada por defecto y no puede ser modificada por los hilos dentro del bucle.
    - No se puede usar break dentro de las iteraciones.
- Barrera implícita al final del for.
- Cláusulas disponibles:
    - shared, private, firstprivate.
    - lastprivate: como private, sólo que la variable original queda con el valor de la última iteración del bucle.
    - reduction: realiza una operación de reducción usando el operador indicado con las múltiples copias de la variable correspondiente.
    - nowait: evita la barra implícita al final del bucle.
    - schedule(política[,chunk]): especifica cómo se distribuyen las iteraciones entre los hilos.
        - static: divide en bloques de chunk iteraciones y las asigna en forma round-robin. Si chunk no se especifica, se dividen las iteraciones en bloques de tamaño aproximado.
        - dynamic: divide en bloques de chunk iteraciones y las asigna bajo demanda. Si chunk no se especifica, las iteraciones son asignadas de a 1.
        - guided: basado en dynamic pero decrementando chunk a medida que avanza el bucle. Cuando chunk=1, el bloque de iteraciones se asigna en forma proporcional a las iteraciones pendiente y los hilos que integran el bucle. Cuando chunk = k > 1, el bloque se asigna de igual manera pero nunca será menor a k.
        - auto: se delega la elección al compilador o al sistema.
        - runtime: la planificación la determina la variable de entorno OMP_SCHEDULE.



Constructor sections

- Divide el trabajo en secciones separadas
- Útil para la distribución de trabajo no-iterativo (paralelismo funcional).
- Cada bloque de código indicado por la directiva section es independiente de los demás y es ejecutado una sóla vez por un único hilo, pudiendo hacerlo en paralelo con el resto de los hilos.
- Existe una barrera implícita al final de sections
- Cláusulas disponibles: shared, private, firstprivate, lastprivate, reduction, nowait

Paralelismo anindado

- OpenMP permite habilitar el uso de paralelismo anidado.
- Se requiere que la variable de entorno OMP_NESTED tenga valor TRUE; si no el código es ejecutado por un único hilo.

Constructor single

- Permite que un bloque de código sea ejecutado por un único hilo dentro de una región paralela
- Es ejecutado por el primer hilo del equipo que llega a ese punto de ejecución; el resto espera al final del bloque (hay una barrera implícita).
- Cláusulas disponibles: private, firstprivate, nowait.

Constructor master

- Como single solo que el bloque es ejecutado por el hilo master y no hay barrera implícita.

Constructor barrier

- Implementa un punto de sincronización global entre todos los hilos de un equipo.

Constructor critical

- Permite implementar regiones críticas en forma sencilla.
- Garantiza que a lo sumo un hilo estará dentro de la sección crítica.
- Si un hilo alcanza un bloque critical y ya hay otro, espera a que la sección crítica se libere.
- Nombre es opcional.

Constructor atomic

- Es una variante de la directiva critical para secciones críticas de una única instrucción.
- Analiza la disponibilidad de instrucciones atómicas por hardware, por lo que podría producir mejor rendimiento que critical. Sin embargo, impone algunas restricciones para su uso que lo hacen poco frecuente.

Constructor ordered

- Útil cuando se necesita que cierto segmento de código se ejecute en el mismo orden en que lo haría la versión secuencial. Se usa con el for.
- Representa un punto de serialización en la ejecución.

Directiva flush

- Representa un punto de sincronización de la memoria:
    - Todas las escrituras pendientes en memoria principal serán asentadas.
    - Todas las lecturas pendientes serán realizadas desde memoria principal.
- No suele ser muy usada ya que muchos de las directivas OpenMP incluyen un flush implícito.

Funciones de librería

- Funciones básicas:
    - void omp_set_num_threads (int num_threads) setea el valor de la variable de entorno OMP_NUM_THREADS.
    - num_threads int omp_get_num_threads () retorna el número de hilos de la región paralela actual.
    - Etc. son muchas.
- Funciones para controlar y monitorizar la creación de hilos.
- Funciones para controlar la planificación de hilos.
- Funciones para el uso de locks.
- Funciones para exclusión mutua recursiva.

Variables de entorno

- OMP_NUM_THREADS: especifica la cantidad de hilos por defecto que se crearán.
- OMP_DYNAMIC: determina si el número de hilos puede ser modificado en forma dinámica.
- OMP_NESTED: especifica si se permite el paralelismo anidado.
- OMP_SCHEDULE: planificación para cuando la cláusula schedule es runtime.

Tasking

- Introducido en la versión 3.0.
- Una tarea es una unidad de trabajo (porción de código) cuya ejecución puede ser diferida en el tiempo.
- Pensado para paralelizar problemas irregulares.
    - Bucles while.
    - Bucles for que no tienen una cantidad conocida de iteraciones.
    - Algoritmos recursivos.
    - Entre otros.
- Cuando un hilo encuentra un constructor task, el sistema de ejecución genera una nueva tarea.
- El momento en que esta tarea se ejecute dependerá del sistema de ejecución
- Se permite el anidamiento de tareas.
- Cláusulas disponibles:
    - shared, private, firstprivate, default
        - Si la cláusula default no fue especificada, entonces:
            - Las variables no especificadas son firstprivate por defecto
            - Las variables que fueron especificadas como shared en la directiva inmediatamente anterior, mantienen su condición.
            - Usar default(none) es recomendable.
            - Las reglas por defecto difieren de la de otros constructores.
    - untied: permite que la tarea pueda ser completada por más de un hilo.
    - if (expresión): evalúa la expresión.
        - Si el resultado es verdadero, se genera una tarea.
        - Si el resultado es falso, se ejecuta el código inmediatamente.
- Barreras para tareas:
    - taskwait: el hilo se suspende hasta que todas sus tareas hijas se hayan completado (solo considera hijas, no descendientes).
- Usarlo para lo que fue pensado, no para paralelismo que OpenMP soporta adecuadamente.
- El rendimiento de estos programas depende del sistema de ejecución.

OpenMP 4.0: Nuevas características

- Soporte para aceleradores, como GPUs y Xeon Phi.
    - Constructores device, host device, target device, etc.
- Soporte para vectorización guiada.
    - Constructor simd.
- Cancelación de hilos.
    - Constructor cancel.
- Afinidad de hilos.
    - Mapeo dinámico de hilos a núcleos.

Clase 5 – Análisis de rendimiento

Métricas - Tiempo de ejecución

- El tiempo de un programa paralelo depende del tamaño de los datos de entrada, del número de procesadores y de los parámetros de comunicación de la arquitectura de soporte.
- Incorrecto analizar el algoritmo paralelo en forma aislada.
    - Se debe realizar a nivel de sistema paralelo (combinación de algoritmo paralelo y contexto de hardware y software).
- Tiempo de ejecución secuencial (Ts): tiempo que tarda en ejecutarse la solución secuencial.
- Tiempo de ejecución paralela (Tp): el tiempo desde que empieza a ejecutarse la primer tarea hasta que termina la última.

Fuentes de overhead

- Existen factores que generan overhead en los programas paralelos e impiden una mejora proporcional al aumento de la arquitectura:
    - Ocio.
    - Interacción entre procesos.
    - Cómputo adicional.

Speedup

$$S_p(n) = \frac{T_s(n)}{T_p(n)}$$

- Refleja el beneficio de usar procesamiento paralelo para resolver un problema en comparación con la usar un procesamiento secuencial.
- Medida de cuantas veces más rápido es el algoritmo paralelo con p unidades de procesamiento comparado al algoritmo secuencial.
- Para computar el Speedup, siempre se debe considerar el mejor algoritmo secuencial (el que resuelva el problema en menos tiempo).
- Límites del Speedup:
    - Si $Sp(n) < 1$ entonces el algoritmo paralelo tarda más que el mejor algoritmo secuencial.
    - El mejor resultado si se distribuye el trabajo entre las unidades de procesamiento sin introducir ocio, interacción ni cómputo adicional → poco usual
    - Con p unidades de procesamiento si $Sp(n) = p$ se tiene Speedup óptimo,
    - Teóricamente, siempre se cumple que $Sp(n) \leq p$.
    - En la práctica, a veces se puede dar $Sp(n) > p$ (fenómeno conocido como Speedup superlineal)
        - Un motivo puede ser que la versión paralela del algoritmo realice menos trabajo que la versión secuencial.
        - Un segundo motivo es la combinación de características de hardware y distribución de los datos del algoritmo paralelo que ponen en desventaja al algoritmo secuencial.

- En arquitecturas heterogéneas, el Speedup se debe calcular considerado la Potencia Cómputo Total (pct) en lugar del número de unidades de procesamiento (p)

$$pct = \sum_{i=0}^{p-1} pcr_i$$

$$pcr_i = \frac{p_i}{p_m}$$

pct	Potencia de Cómputo Total
pcr	Potencia de Cómputo Relativa
p_i	Potencia del procesador *i*
p_m	Potencia del mejor procesador

Eficiencia

$$E_p(n) = \frac{S_p(n)}{S_{Opt}}$$

- Permite saber qué tan cerca se está del speedup óptimo, que tan bien se está aprovechando la arquitectura. Se calcula de la siguiente manera:
- En arquitecturas homogéneas $SOpt = p$ mientras que en heterogéneas $SOpt = pct$
- Si $Sp(n) = p$ (sistema paralelo ideal), entonces $Ep(n) = 1$
- En la práctica, $Sp(n) \leq p$ lo que implica que $Ep(n) \leq 1$
- Por definición, $Ep(n) > 0$. Por lo tanto, $0 < Ep(n) \leq 1$

Overhead total

$$OT_p(n) = pT_p(n) - T_s(n)$$

- Es la diferencia entre la suma del tiempo requerido por todas las unidades de procesamiento y el del mejor algoritmo secuencial para resolver el mismo problema empleando una única de unidad de procesamiento.

Overhead de las comunicaciones

$$OC_p(n) = \frac{Tcomm_p(n)}{T_p(n)} \times 100$$

- La relación entre el tiempo requerido por las comunicaciones de la solución y el tiempo total que esta requiere.

Escalabilidad

- Hace referencia a la capacidad que tiene un sistema de mantener un nivel de Eficiencia fijo al incrementar tanto el número de unidades de procesamiento como el tamaño del problema de resolver. Se dice que el sistema es **escalable**
- Es una medida de la capacidad del sistema de incrementar el Speedup en forma proporcional al número de unidades de procesamiento empleadas.

- Escalabilidad fuerte: cuando al incrementar el número de unidades de procesamiento, no resulta necesario aumentar el tamaño de problema para mantener la eficiencia en un valor fijo.
- Escalabilidad débil: Cuando al incrementar el número de unidades de procesamiento, resulta necesario también aumentar el tamaño de problema para mantener la eficiencia en un valor fijo.
- Agregar más procesadores no suele ser una solución directa.
- Los algoritmos pueden tener límites inherentes para su escalabilidad.
- Los recursos de hardware desempeñan un factor fundamental en la escalabilidad.

Ley de Amdahl (para escalabilidad fuerte)

- Los factores de overhead limitan los beneficios del procesamiento paralelo.
- Una restricción importante proviene de aquellos secciones de código que no pueden ser paralelizadas.
- La Ley de Amdahl permite estimar el Speedup alcanzable en aquellos programas paralelos que contienen partes secuenciales.
- El Speedup ahora puede reescribirse de la siguiente forma:
    - Dada una fracción f, $0 \leq f \leq 1$, de un programa paralelo que debe ser ejecutada secuencialmente:

$$S^A{}_p\,(n) = \frac{1}{f + \frac{(1-f)}{p}}$$

- El escalado perfecto se logra cuando el problema se resuelve en 1/P unidades de tiempo (comparado al secuencial)

Ley de Gustafson (para escalabilidad débil)

- El incremento en el Speedup por un tamaño mayor de problema no es percibido por la Ley de Amdahl
- Gustafson reescribió la ecuación para estimar el máximo speedup alcanzable (conocido como Speedup escalado)
    - Dada una fracción f', $0 \leq f' \leq 1$, de un programa paralelo que debe ser ejecutada secuencialmente pero que no crece en forma proporcional al tamaño de problema, el Speedup escalado se calcula como:

$$S^S{}_p(n) = \frac{T_s(n)}{T_p(n)} \quad p + (1-p) \times f'$$

- El escalado perfecto se logra cuando se resuelve un problema P veces más grande en la misma cantidad de tiempo que el secuencial.

Desbalance de carga

- En arquitecturas heterogéneas es útil poder medir el Desbalance de carga: D.
- Si todas las unidades de procesamiento toman el mismo tiempo, entonces $D = 0$
- En general, se debe intentar que D esté lo más cerca posible de 0.

Recomendaciones para medir tiempos de ejecución

- En la práctica, el tiempo de ejecución no siempre se considera desde el que programa empieza hasta que el mismo termina.
- Tp(n) debe ser un único valor que contemple el tiempo que transcurre desde que la primera tarea comenzó a ejecutar hasta que la última haya completado su trabajo, este tiempo puede hacer referencia a una determinada parte del programa.
- Para asegurar una medición correcta, puede ser útil emplear barreras.
- Precisión en medición.
- Tener en cuenta es la variabilidad en las mediciones:
    - Repetir pruebas, calcular promedio o mediana.
    - Reportar mínimos y máximos.

Clase 6 – Diseño de algoritmos paralelos

Diseño de algoritmos paralelos

- Pasos fundamentales: Descomposición en tareas y Mapeo de tareas a procesos.

Descomposición en tareas

- Proceso de dividir el cómputo en partes más pequeñas (Tareas), de las cuales algunas o todas podrán ser potencialmente ejecutadas en paralelo.
- Hay que definir un gran número de tareas pequeñas para obtener una descomposición de grano fino. Seguramente en etapas posteriores se descartan algunas opciones de descomposición consideradas inicialmente, aglomerando tareas.
- Se puede realizar de diferentes modos:
    - Descomposición de datos:
        - Descomponer los datos en pequeñas porciones y luego asociar el cómputo relacionado a las porciones para generar las tareas.
        - Esto lleva a un número determinado de tareas.
        - Una operación puede requerir datos de diferentes tareas, lo que lleva a comunicación y sincronización.
        - Son posibles diferentes particiones.
        - Mas usada.
    - Descomposición funcional:
        - Se enfoca en el cómputo a realizar más que en los datos.
        - Divide al cómputo en tareas disjuntas y luego examina los datos.
        - Los requerimientos de datos pueden ser disjuntos o superponerse significativamente.
        - Es una forma diferente de pensar los problemas.
- Si el problema lo permite, todas las tareas serán independientes. Esto no es lo usual y existe algún tipo de dependencia entre las tareas.
    - Un Grafo de Dependencias de Tareas (GDT) puede ser útil para expresar las dependencias entre las tareas y su orden relativo

Granularidad de las tareas

- Grano fino: gran número de pequeñas tareas
- Grano grueso: pequeño número de grandes tareas

Grado de concurrencia

- Indicado por el número de tareas que se ejecutan en paralelo
- Interesante a conocer es el máximo grado de concurrencia alcanzable por una descomposición. Es más útil conocer el grado de concurrencia promedio.
- El camino crítico del grafo determina el grado de concurrencia promedio para una determinada granularidad.
    - Camino dirigido más largo entre un nodo inicial y un nodo final.
    - La suma de los pesos de los nodos que integran el camino crítico se conoce como longitud del camino crítico
    - Grado de concurrencia promedio = Peso total / Longitud del camino crítico.
    - Camino corto → mayor grado de concurrencia.
- Puede parecer que el tiempo de ejecución paralela se puede reducir indefinidamente con una descomposición cada vez más fina, pero esto no es posible ya que en general, hay un límite inherente al problema sobre qué tan fina puede ser una descomposición. Además se debe tener en cuenta que las tareas deben comunicarse y sincronizar.
    - Un adecuado balance entre cómputo y comunicación definirá el rendimiento alcanzable

Aglomeración de tareas

- Consiste en analizar si conviene combinar varias tareas para obtener un número de tareas menor pero de mayor tamaño. También se analiza si vale la pena replicar datos o cómputo.
- El número final de tareas como resultado debe ser igual al numero de procesadores.
- Hay 3 objetivos que guían las decisiones de aglomeración y replicación:
    - Incremento de la granularidad: al combinar varias tareas relacionadas, se elimina la necesidad de comunicar datos entre ellas.
    - Preservación de la flexibilidad: al combinar varias tareas se puede limitar la escalabilidad y portabilidad del algoritmo.
    - Reducción de costos de desarrollo: a veces el costo puede ser muy elevado para la ganancia asociada.

Técnicas de descomposición

- Recursiva:
    - Se ajusta bien a problemas que se pueden resolver mediante divide y vencerás.
    - El problema inicial es dividido en un conjunto de subproblemas independientes. Luego, cada uno de estos subproblemas son recursivamente descompuestos en otros subproblemas independientes más pequeños hasta alcanzar una determinada granularidad.
    - Puede requerirse alguna fase de combinación de resultados parciales.
- Basada en los datos:
    - Usada en problemas que operan sobre grandes estructuras de datos.
    - Dos pasos:
        - Particionar los datos que se procesarán.

        - Usar la partición anterior para descomponer el cómputo en tareas.
    - Se puede realizar de diferentes maneras
        - De salida:
            - Cuando cada elemento de la salida de un programa se puede calcular en forma independiente como función de los datos de entrada.
            - Una determinada descomposición de datos lleva a una dada descomposición del cómputo en tareas, pero puede haber más de una opción.
        - De entrada:
            - Particionar los datos de salida no siempre es posible por lo que se particionan los datos de entrada.
            - A cada tarea se le asigna una porción de los datos de entrada. En ocasiones, se puede requerir de algún paso posterior de reducción de salidas parciales.
        - Intermedios:
            - Los algoritmos pueden ser estructurados en múltiples etapas de forma tal que la salida de una etapa es la entrada de la siguiente.
- Exploratoria:
    - Se emplea en aquellos problemas cuya solución involucra una búsqueda en un espacio de soluciones
    - Se particiona el espacio de búsqueda en porciones más pequeñas y se realiza una búsqueda concurrente en cada una de ellas hasta encontrar la solución objetivo.
    - Diferencia con descomposición basada en los datos: las tareas son ejecutadas completamente, en exploratoria no. Como alguna de las configuraciones está más cerca de la solución se avanza en esas.
- Especulativa:
    - Se usa cuando un programa podría tomar uno de varios caminos que implican cómputo significativo pero la decisión depende de la salida de algún cómputo anterior.
    - Como un case con múltiples opciones que son evaluadas al mismo tiempo pero antes de tener el valor de la entrada. Cuando la entrada del case está disponible, se descartan las opciones incorrectas y se continúa la ejecución.
- Híbrida: combinación de las anteriores.

Mapeo de tareas a procesos

- Asignar las tareas a los procesos del programa.
- Hay que tener en cuenta las características de las tareas:
    - Modo de generación
        - Estática: las tareas que se generan se conocen previo a la ejecución
        - Dinámica: las tareas se generan durante la ejecución, por lo que no se conoce de antemano la cantidad total.
    - Tamaño y conocimiento de este:

        - Tareas uniformes: las tareas requieren aproximadamente el mismo tiempo de cómputo.
        - Tareas no uniformes: el tiempo requerido entre una tarea y otra puede variar significativamente.
        - Conocer el tamaño de las tareas previo a la ejecución es otro factor que puede influir en el mapeo
    - Volumen de datos asociado:
        - Tiene que ver muchas veces con la granularidad elegida.
        - La granularidad impacta directamente en la relación cómputo-comunicación.
            - Con bajos niveles de comunicación se tiende a reducir la granularidad y asignar un menor volumen de datos por proceso.
            - Con mucho intercambio se suele optar por aumentar la granularidad o emplear memoria compartida.

Técnicas de mapeo

- Se mapea teniendo en cuenta que el tiempo para completar las tareas debe ser el mínimo posible. Dos estrategias en conflicto que deben balancearse:
    - Asignar tareas independientes en diferentes procesadores para lograr un mayor grado de concurrencia.
    - Asignar tareas que se comunican frecuentemente en el mismo procesador reducir overhead y mejorar localidad.
- Para mapear entonces se debe tener en cuenta el grafo de dependencias de tareas y la interacción entre las mismas:
    - Las dependencias pueden condicionar el balance de carga entre los procesos.
        - Carga balanceada no necesariamente significa mínimo tiempo de ejecución.
    - La interacción debe tender a minimizar la comunicación entre los procesos.

Técnicas de mapeo para el balance de carga

- Se pueden clasificar en:
    - Estáticas:
        - Distribuyen las tareas entre los procesos previo a la ejecución.
        - Es fundamental conocer las características de las tareas.
        - Para casos complejos se emplean heurísticas
        - Los algoritmos son más fáciles de diseñar y programar.
        - Suele ser utilizado en problemas que emplean descomposición basada en los datos.
        - Como las tareas están fuertemente relacionadas con los datos, mapear los datos a los procesos es de alguna forma equivalente a mapear las tareas a los procesos.
    - Dinámicas:
        - Distribuyen las tareas entre los procesos durante la ejecución.
        - Deben mapearse dinámicamente.

- El mejor si no se conoce de antemano el tamaño de las tareas.
- Podría generar un alto overhead por la migración de datos.
- Necesario cuando:
    - Estático lleva a desbalanceo
    - El grafo de dependencias de tareas es dinámico
- Se suele referir a sus técnicas como balance de carga dinámico
- Los esquemas de mapeo dinámico se clasifican en:
    - Centralizados:
        - Proceso masters que administra las tareas a realizar.
        - El resto son worker.
        - Cuando un proceso worker no tiene trabajo, le pide al master que le asigne una tarea y así hasta que no hallan tareas.
        - Fácil de implementar pero sufren de escalabilidad limitada: el master se puede volver un cuello de botella cuando la cantidad de procesos es muy grande.
    - Distribuidos:
        - Se evita el cuello de botella potencial del master, delegando la distribución entre varios procesos pares.
        - Más difícil de implementar.
        - Los problemas que surgen son de sincronización.

Métodos para reducir el Overhead de las interacciones

- Minimizar volumen de datos intercambiados: a mayor volumen de datos intercambiados, mayor tiempo de comunicación.
- Minimizar frecuencia de las interacciones: conviene combinar varias comunicaciones en una sola.
- Minimizar competencia entre recursos y zonas críticas: evitar posibles cuellos de botella mediante el uso de técnica descentralizadas.
- Solapar cómputo con comunicaciones.
- Replicar datos o cómputo: si permite reducir las interacciones.
- Usar operaciones de comunicación colectiva.
- Solapar comunicaciones con otras comunicaciones: siempre que sea soportado.

Modelos de Algoritmos Paralelos

- Representa una estructura usual de código que combina técnicas de descomposición de problema y de mapeo de tareas junto a la aplicación de métodos para minimizar overhead.

Maestro-Esclavo

- El proceso Maestro es el responsable de generar trabajo y asignárselo a los Workers.
- Dos opciones de distribución de trabajo:

    - Mapeo estático si el Maestro puede estimar de antemano el tamaño de las tareas.
    - Mapeo dinámico en donde tareas pequeñas son asignadas a los workers en múltiples instancias.
    - Master puede convertirse en cuello de botella si las tareas son muy pequeñas o los workers son muy rápidos.
    - Puede ser generalizado a múltiples niveles.
    - Resulta adecuado tanto para memoria compartida como para pasaje de mensajes.

Pipeline

- El cómputo se descompone en una secuencia de procesos.
- Los datos son particionados y pasados entre los procesos que realizan una tarea sobre ellos.
- Normalmente se organizan en forma de arreglo lineal o multidimensional, menos comunes son árboles o grafos.
- Puede ser visto como una cadena de productores y consumidores.
- El balance de carga depende de la granularidad de las tareas:
    - A mayor granularidad, más tiempo tardará el pipeline en llenarse.
    - A menor granularidad, mayor interacción entre los procesos del pipeline.

Single Program Multiple Data (SPMD)

- Cada proceso realiza el mismo cómputo sobre una porción de datos diferentes
    - Los procesos pueden tomar diferentes caminos
- La carga de trabajo es proporcional a la cantidad de datos asignados a un proceso.
    - Dificultades en problemas irregulares.
- El cómputo puede involucrar diferentes fases, las cuales son usualmente intercaladas con comunicación/sincronización.
- Adecuado tanto en memoria compartida como en pasaje de mensajes
    - En memoria compartida, el esfuerzo de programación suele ser menor.
    - En pasaje de mensajes:
        - Cuando el espacio de direcciones está particionado se tiene mayor localidad de datos
        - El overhead de las comunicaciones se puede reducir usando comunicaciones no bloqueantes.

Divide y Vencerás

- Dos fases:
    - Dividir: fase en la que se particiona sucesivamente el problema en subproblemas más pequeños hasta obtener una granularidad deseada.
    - Conquistar: fase en la que se resuelven los subproblemas en forma independiente.
    - A veces se requiere una fase adicional de combinación de resultados parciales para llegar al resultado final.

Clase 7 – Programación en pasaje de mensajes – Estándar MPI

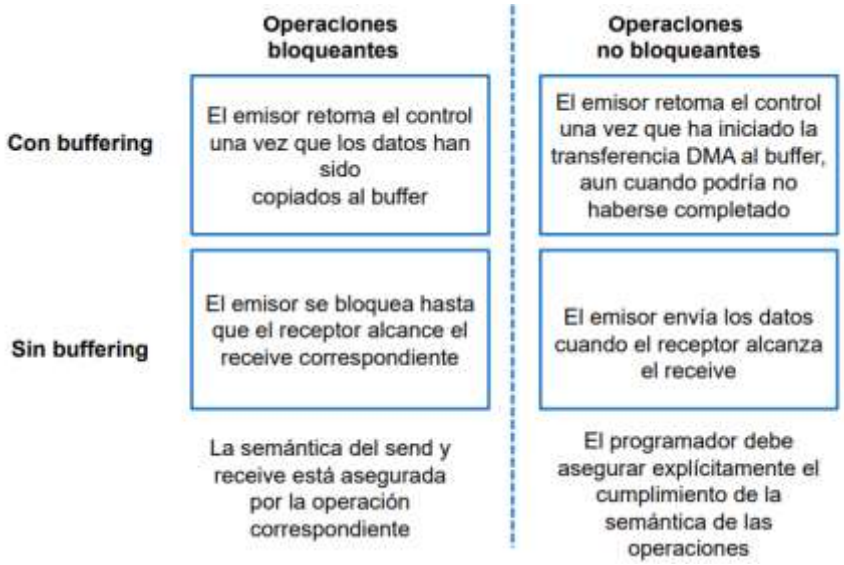
Fundamentos del modelo de pasaje de mensajes

- Programabilidad:
  - Se trabaja en bajo nivel.
  - Programador responsable de la distribución de datos y el mapeo de procesos, y la comunicación.
  - Difícil de programar, depurar y mantener.
  - Se suele escribir siguiendo el modelo SPMD.
    - No todos terminan ejecutando lo mismo
    - Los procesos no están sincronizados en la ejecución de cada sentencia.
- Eficiencia → Ajuste para mejorar el rendimiento puede ser óptimo.
  - Se puede manejar el balance de carga, la redistribución de datos y procesos, replicar datos, etc.
- Portabilidad → Existen librerías para facilitar la ejecución de código sobre diferentes arquitecturas.
  - No garantiza portabilidad de rendimiento.

Operaciones Send y Receive

- Existe diferentes protocolos para Send y Receive:
  - Bloqueante:
    - Devuelve el control al proceso llamador cuando los recursos involucrados puedan ser reutilizados.
    - Garantiza que todas transiciones de estados iniciadas por la operación fueron completadas.
    - Ociosidad en los procesos
    - Sin buffering:
      - El send se bloquea hasta que el receptor no termine el receive del mensaje.
        - Tiempo ocioso de los procesadores.
        - Deadlocks si las sentencias de comunicación no coinciden.
    - Con buffering:
      - El send se bloquea hasta que el mensaje llega a un buffer prealocado del sistema (diferente al del receptor).
        - Transmisión del mensaje:
          - Hardware para comunicación asincrónica (sin intervención de la CPU): se comienza la transmisión al buffer del receptor.
          - Sin hardware especial: el emisor trasmite el mensaje al buffer del receptor y recién ahí se desbloquea.
        - Reducen el tiempo ocioso pero aumentan el costo por manejo de buffers.

        - Reduce la ocurrencia de deadlocks pero no los evita.
    - No bloqueante:
        - Para evitar overhead se devuelve el control de la operación inmediatamente.
            - No garantiza que los recursos puedan ser reutilizados.
            - No garantiza que todas las operaciones hayan sido completadas.
        - Requiere un posterior chequeo para asegurar la finalización de la comunicación.
        - Hay dos alternativas:
            - Sin buffering: inicia comunicación al llegar al receive.
            - Con buffering: el emisor utiliza acceso directo a memoria para copiar los datos a un buffer prealocado mientras el proceso continúa su cómputo
                - Reduce el tiempo en que el dato no está seguro.



Estándar MPI

- En los 90s existían muchas librerías de pasaje de mensajes no compatibles.
- MPI define una librería estándar que puede ser empleada desde C hasta otros lenguajes.
- Muchas implementaciones de MPI hoy día.
- Modelo SPMD
- Define muchísimas rutinas pero tan solo con 6 se pueden escribir programas paralelos:
    - **MPI_Init:** inicializa el entorno MPI. Debe ser invocada por todos los procesos como primer llamado a rutina MPI.
    - **MPI_Finalize:** cierra el entorno MPI. Debe ser invocado por todos los proceso como ultimo llamado a rutina MPI.
    - Comunicador define el dominio de comunicación
        - Variables del tipo MPI_Comm almacenan información sobre los procesos que pertenecen a él.

        - Un proceso puede pertenecer a muchos comunicadores.
        - Comunicador con todos los procesos: MPI_COMM_WORLD.
        - En cada operación de comunicación hay que indicar comunicador
        - **MPI_Comm_size:** indica la cantidad de procesos en el comunicador.
        - **MPI_Comm_rank:** indica el id del proceso dentro de ese comunicador.
            - Cada proceso puede tener un rank diferente en cada comunicador.
    - **MPI_Send:** rutina básica para enviar datos a otro proceso.
    - **MPI_Recv:** rutina básica para recibir datos de otro proceso

Operaciones de comunicación

- MPI soporta:
    - Comunicaciones punto a punto: involucran a dos procesos, pueden ser bloqueantes y no bloqueantes.
        - En las bloqueantes tener mucho cuidado con el deadlock.
    - Comunicaciones colectivas: involucran dos o más procesos, pueden ser bloqueantes y no bloqueantes.
- Diferentes variantes para MPI_Send (bloqueantes):
    - MPI_Send:
        - Retorna el control sólo cuando el buffer del emisor está listo para ser reusado. No significa que el receptor ya haya recibido.
        - Podría involucrar uso de buffering o no.
    - MPI_Bsend (Buffered send):
        - Permite implementar buffering a nivel de usuario.
    - MPI_Ssend (Synchronic Send).
        - Retorna el control sólo cuando el buffer del emisor está listo para ser reusado y el proceso receptor ha comenzado a recibir el mensaje.
    - MPI_Rsend (Ready Send).
        - Sólo puede ser invocado si el proceso receptor se encuentra listo para recibir.
        - Mejorar el rendimiento de las comunicaciones pero es más inseguro.
        - Prácticamente no se utiliza por sus restricciones.

Comunicaciones punto a punto no bloqueantes

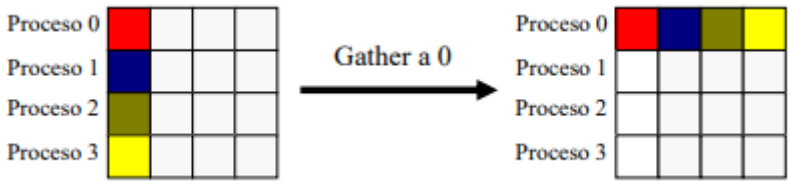
- Comienzan la operación de comunicación e inmediatamente devuelven el control (no garantiza que la operación haya finalizado).
    - MPI_Isend y MPI_Irecv.
    - Permite solapar cómputo con comunicación y es responsabilidad del programador que no haya errores.
- MPI_Test: evalúa si la operación de comunicación finalizó.
- MPI_Wait: bloquea al proceso hasta que la operación indicada en el Request haya finalizado.

Orden y fairness

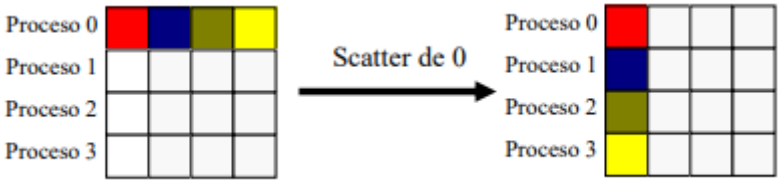
- MPI asegura que los mensajes no se sobrepasarán entre ellos.
    - Si un proceso envía mensajes M1 y M2 al receptor y ambos coinciden con el mismo receive, el orden de recepción será: M1, M2.
    - Si un proceso hace recepciones R1 y R2 y hay un mensaje pendiente que coincide con ambos, R1 recibirá antes que R2.
- MPI no asegura fairness. Es responsabilidad del programador.
- Es posible consultar si hay comunicaciones pendiente y algunos de sus datos con MPI_Probe o su alternativa no bloqueante MPI_Iprobe.

Comunicaciones colectivas

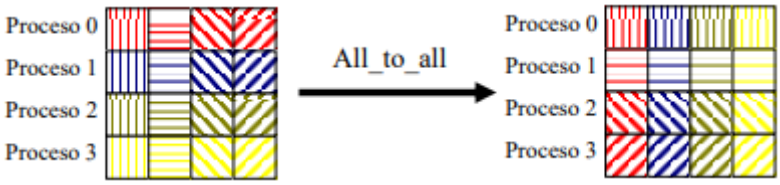
- Las funciones para operaciones de comunicación colectiva de MPI no sólo facilitan la programación sino que mejoran el rendimiento.
- Todos los procesos del comunicador deben llamar a la rutina colectiva.
- Hay tres tipos:
    - Sincronización: los procesos se bloquean hasta que todos hayan llegado a determinado punto del programa.
        - Sincronización en una barrera con MPI_Barrier.
    - Transferencia de datos.
        - MPI_Bcast: un proceso envía el mismo mensaje a todos los otros procesos del comunicador.
        - MPI_Gather: recolecta un vector de datos de cada proceso del comunicador y los concatena en orden para dejar el resultado en un único proceso.



            - MPI_Gatherv para que puedan enviar una cantidad de datos diferentes.
            - Allgather para que el resultado sea enviado a todos los procesos.
            - Allgatherv combinación de los de arriba.
        - MPI_Scatter: reparte un vector de datos entre todos los procesos de forma equitativa.

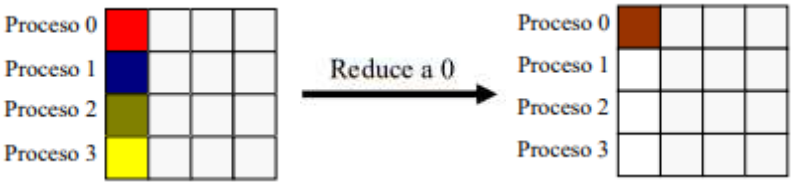


            - Scatterv para repartid cantidades diferentes.
        - MPI_Alltoall: cada proceso envía una parte de sus datos a cada uno de los otros procesos y recibe de ellos una parte.
            - Todas las porciones son del mismo tamaño.
            - Es equivalente a realizar un Scatter + Gather



- Alltoallv para enviar y recibir porciones de tamaño diferentes

- Computaciones colectivas: operaciones de reducción:
  - MPI_Reduce: combina los elementos enviados por cada uno de los procesos aplicando una cierta operación.



- MPI_Allreduce para enviar el resultado a todos los procesos.

- A veces las comunicaciones colectivas se realizan entre subconjuntos de un comunicador, se usa MPI_Comm_split para dividir el comunicador en subgrupos.

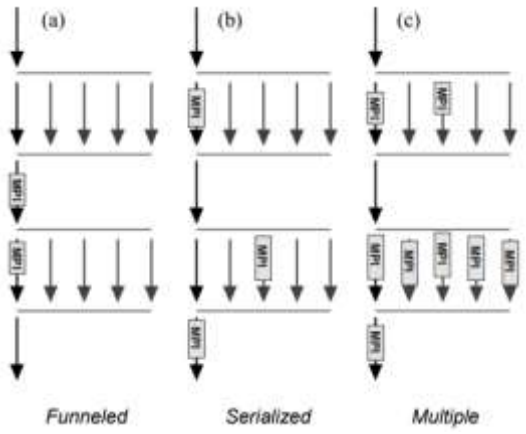
Clase 8 – Programación híbrida

- Modelo híbrido explota mejor las características de los clusters multicore:
  - Las tareas en el mismo nodo se comunican y sincronizan por memoria compartida.
  - Las tareas en diferentes nodos se comunican y sincronizan por pasaje de mensajes.
  - Combinar MPI con OpenMP o Pthreads.
- El modelo de programación híbrido puede incrementar (no siempre) el rendimiento y la escalabilidad de una aplicación,
- Razones para utilizar el modelo híbrido:
  - Aprovechar la memoria compartida dentro de cada nodo:
    - Reduce overhead de las comunicaciones MPI
    - Reducen requerimientos de memoria de la aplicación.
  - Algunas aplicaciones presentan dos niveles de paralelismo:
    - Paralelismo de grano grueso: gran cantidad de cómputo que puede ser realizado en forma independiente + algún intercambio de información ocasional entre los procesos de la aplicación (MPI)
    - Paralelismo de grano fino, disponible a nivel de bucle (OpenMP)
  - Algunas aplicaciones presentan una carga de trabajo desbalanceada al nivel de MPI, la cual puede resultar difícil de equilibrar
    - Balancear la carga en forma dinámica con OpenMP resulta más sencillo de lograr.
- Razones para no utilizar el modelo híbrido:
  - Algunas aplicaciones sólo presentan un único nivel de paralelismo.
  - Al introducir OpenMP o Pthreads a un código MPI existente también se están introduciendo sus desventajas.

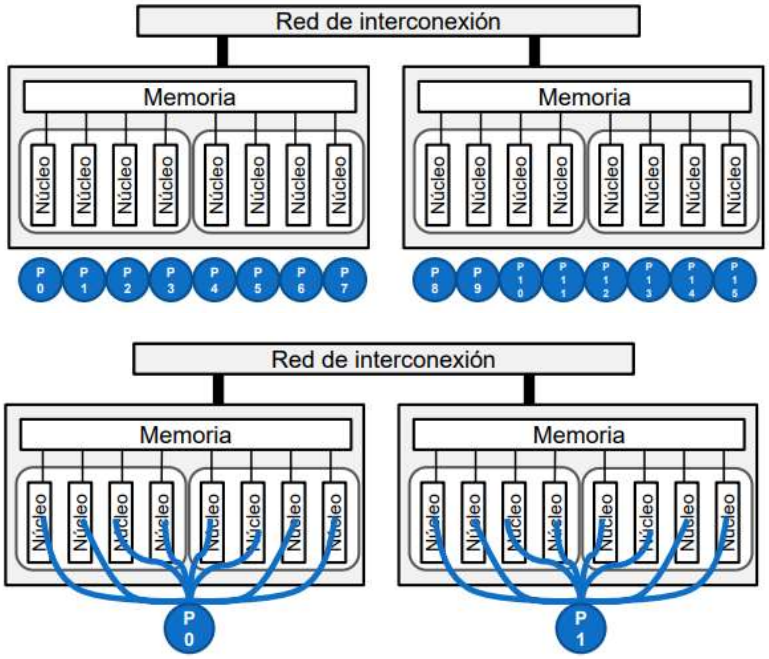
- Existen diferentes esquemas para paralelizar una aplicación utilizando el modelo híbrido.
    - Sin solapamiento de cómputo y comunicaciones:
        - Conocido como master-only o modo vector.
        - Proceso MPI por nodo y OpenMP (o Pthreads) sobre los núcleos de los nodos.
        - Llamadas a rutinas MPI fuera de las regiones paralelas o código de hilos.
        - Ventajas:
            - No hay intercambio de mensajes dentro de cada nodo
            - La topología de los procesos MPI ya no es relevante para optimizar rendimiento.
        - Desventajas:
            - Hilos ociosos mientras el master comunica.
            - Un único hilo seguramente no aprovecha todo el ancho de banda.
    - Con solapamiento de cómputo y comunicaciones
        - Para evitar el ocio de hilos permitiendo que más de un hilo pueda comunicarse con otro.
        - Ventajas:
            - Se reduce el tiempo ocioso.
            - Se aprovecha el ancho de banda
        - Desventajas:
            - Mas complejo de programar
            - Hay que equilibrar la carga de trabajo entre los hilos que comunican y los que no lo hacen.

Soporte MPI para programación híbrida

- MPI especifica 4 niveles diferentes:
    - MPI_THREAD_SINGLE (Nivel 0): Sin soporte para hilos
    - MPI_THREAD_FUNNELED (Nivel 1): Los procesos pueden ser multihilados pero todas las comunicaciones las realizará el hilo master
    - MPI_THREAD_SERIALIZED (Nivel 2): Los procesos pueden ser multihilados y los diferentes hilos pueden ejecutar rutinas MPI pero sólo una a la vez.
    - MPI_THREAD_MULTIPLE (Nivel 3): Múltiples hilos pueden realizar múltiples comunicaciones, sin restricciones.



- MPI_Init debe reemplazarse por MPI_Init_thread.
- MPI vs MPI con OpenMP.



Clase 9 – Tendencias en HPC

- El objetivo de HPC ha sido incrementar el rendimiento y ocasionalmente el cociente precio/rendimiento
- TOP500: ranking que lista las 500 supercomputadoras más potentes del mundo.
    - Para el cálculo de la potencia se emplea un benchmark específico llamado LINPACK.
    - GFlops
- GREEN500: ranking que lista las 500 supercomputadoras más eficientes desde el punto de vista energético del mundo.
    - Para el cálculo de la potencia se emplea un benchmark específico llamado LINPACK y además se mide consumo energético
    - GFlops sobre Watts
- Hubo una reciente consolidación de aceleradores en HPC.
    - Acelerador: dispositivo de hardware diseñado para mejorar el rendimiento del sistema (GPUs o coprocesadores)
    - Son más eficientes energéticamente.
    - Aumenta costos de programación y complejidad
    - Complica mantenimiento y extensión de código a futuro
- ¿Hay supercomputadoras en Argentina?
    - Si.

GPUs

- Diseñadas para procesamiento de gráficos. Por la potencia de cálculo, se comenzó a aumentar su grado de programación, permitiendo que se usen para resolver problemas de propósito general.
- Arquitecturas de memoria compartida, inspiradas en el modelo SIMD de Flynn.

- Se adaptan mejor para aplicaciones que admiten paralelismo de datos.
- Tienen una jerarquía de memoria compleja:
    - Memoria global: sirve de memoria principal.
    - Memoria compartida: administrada por software y accesible por todos los hilos del multiprocesador.
    - Memoria de constantes: de solo lectura, esta en la memoria principal y es visible por todos los hilos.
    - Memoria de texturas: parecida a la constantes y es optimizada para localidad especial 2D.
- CUDA estándar para la programación.
- OpenCL Estándar para programación paralela multiplataforma
    - Se lo puede ver como una versión de CUDA generalizada.
- SYCL evolución de OpenCL.

Memoria de Alto Ancho de Banda (HBM)

- Nuevo tipo de memoria RAM.
- Organiza los chips de memoria en forma vertical y apilada.
- Ahorro de espacio y considerables aumentos en la velocidad de comunicación.

Resumen de aceleradores

- Aspectos que son comunes a todos ellos:
    - Complejidad de programación
        - Técnicas de programación y optimización específicas, múltiples niveles de paralelismo, balance de carga, diversidad de modelos de lenguajes/modelos de programación, ausencia de estándar (consolidado).
    - Memoria del dispositivo separada
        - Al estar separada de la memoria del host, su administración es clave para obtener alto rendimiento.
        - Recientemente se han desarrollado modelos de memoria unificada.
    - Patrón de acceso a la memoria
        - Requieren de patrones de acceso específicos para obtener el mejor rendimiento, que no siempre coinciden con los de las CPU.
    - Manycores
        - Gran cantidad de núcleos pequeños.
    - Multihilado
        - Aplicado de diferentes maneras, buscan ocultar la latencia de la memoria.
    - Vectorización (SIMD).
    - Ejecución de instrucciones en orden.
        - La lógica de control está simplificada.
    - Memorias caché más pequeñas .
        - La mayor parte de los recursos se destina a unidades funcionales.
        - Cobra mayor importancia explotar localidad de datos.