Pthreads

Sistemas Operativos y lenguajes proveen mecanismos para permitir la programación de aplicaciones “multithreading”.

En los 90s se desarrollo una biblioteca en C, Pthreads, para programación paralela en memoria compartida en la cual se pueden crear threads, asignarles atributos, darlos por terminados, identificarlos, etc.

Hay varias APIs para el manejo de Threads, Pthreads ha emergido como API estándar para el manejo de Threads.

Trabaja con funciones reentrantes, que son aquellas funciones que pueden ser llamadas por mas de un proceso al mismo tiempo sin inconvenientes.

Puntos importantes del funcionamiento de Pthreads

- El programa principal termina cuando terminan todos los hilos.
- Funciones basicas
    - int pthread_create (pthread_t *thread_handle, const pthread_attr_t *attribute, void * (*thread_function)(void *), void *arg);
        - Genera hilo
        - Argumentos: 1ro el manejador, si se quiere referirse al hilo se refiere al manejador; 2do ni idea; 3ra la función que se ejecutara en el hilo; 4ta parámetros para la función, se recomienda no pasar algo que se modificara.
    - int pthread_exit (void *res);
        - Termina ejecución hilo. Debe estar al final de la función que se pasa como 3er parámetro en el créate.
    - int pthread_join (pthread_t thread, void **ptr);
        - Espera que el hilo termine de ejecutarse
    - int pthread_cancel (pthread_t thread);
        - Solicita que se termine el hilo

Exclusion mutua

- Secciones criticas implementadas haciendo uso del bloqueo por exclusión mutua en variables mutex.
- Estas variables están en el programa principal.
- 2 operaciones: locked y unlocked.
- Solo un hilo puede bloquear una variable y esta solo puede ser desbloqueadas por quien la bloquea, es por ello que todos los mutex deben iniciarse desbloqueados.
- Básicamente para entrar a una sección critica un Thread debe lograr tener control del mutex
- Funciones
    - int pthread_mutex_lock (pthread_mutex_t *mutex);
        - Bloquear
    - int pthread_mutex_unlock (pthread_mutex_t *mutex);
        - Desbloquear

- int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *lock_attr);
    - Inicializar la variable. Hay varios tipos de mutex (locks) que se especifican en el primer parámetro:
        - Normal: Un Mutex con el atributo Normal NO permite que un thread que lo tiene bloqueado vuelva a hacer un lock sobre él (deadlock).
        - Recursive: Un Mutex con el atributo Normal SI permite que un thread que lo tiene bloqueado vuelva a hacer un lock sobre él (deadlock).
        - Error Check: Trabaja como el normal, solo que en lugar de dejar bloqueado al proceso, se informa error.

- Para evitar tiempos ociosos se puede utilizar la función pthread_mutex_trylock. Que retorna el control informando si pudo hacer o no el lock.
    - Es no bloqueante: si un mutex esta desbloqueado, lo bloquea e informa que se puedo realizar el bloqueo; si un mutex esta bloqueado, informa que no se puedo hacer el bloqueo y retorna el control.

Ejemplo:

Un thread productor no debe sobrescribir el buffer compartido cuando el elemento anterior no ha sido tomado por un thread consumidor. Un thread consumidor no puede tomar nada de la estructura compartida hasta no estar seguro de que se ha producido algo anteriormente. Los consumidores deben excluirse entre sí. Los productores deben excluirse entre sí.

```
pthread_mutex_t    mutex;
int hayElemento;
tipo_elemento Buffer;
...
main()
  { hayElemento= 0;
    pthread_init ();
    pthread_mutex_init(&mutex, NULL);

     /* Create y join de threads productores y consumidores*/
  }
```

```
void *productor (void *datos)
  {  tipo_elemento elem;
     int ok;
     ....
     while (true)
           { ok = 0;
             generar_elemento(&elem);
             while (ok = = 0)
               {  pthread_mutex_lock(&mutex);
                  if (hayElemento== 0)
                    {  Buffer = elem;
                       hayElemento = 1;
                       ok = 1;
                     }
                  pthread_mutex_unlock(&mutex);
               }
           }
  }
```

```
void *consumidor(void *datos)
  {  int ok;
     tipo_elemento elem;
     ....
     while (true)
        { ok = 0;
          while (ok == 0)
              {  pthread_mutex_lock(&mutex);
                 if (hayElemento == 1)
                    { elem = Buffer;
                      hayElemento = 0;
                      ok = 1;
                     }
                 pthread_mutex_unlock(&mutex);
              }
           procesar_elemento(elem);
        }
   }
```

Variables Condición

- Para que un thread se autobloquee hasta que se alcance un estado determinado del programa.
- Similar a monitores.
- Variable de condición asociada con predicado. Una variable de condición puede asociarse a varios predicados. Si es TRUE la variable de condición da una señal (signal) para los threads que están esperando por el cambio de estado de la condición.
- La variable de condición debe tener siempre un mutex asociada a ella. Siempre debe bloquearse el mutex antes de testear el predicado (antes de dormirme básicamente)

- Si el predicado es falso, el thread espera en la variable condición utilizando la función pthread_cond_wait (NO USA CPU).
- Funciones
    - int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex)
        - El hilo se duerme si la variable es falsa, continua si es verdadera.
        - Internamente como funciona: el hilo desbloquea el mutex y luego se duerme
    - int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mutex const struct timespec *abstime)
        - El hilo se duerme si la variable es falsa por un determinado tiempo, continua si es verdadera.
    - int pthread_cond_signal (pthread_cond_t *cond)
        - Despierta al primero dormido. Cuando se despierta bloquea al mutex.
    - int pthread_cond_broadcast (pthread_cond_t *cond)
        - Despierta a todos
    - int pthread_cond_init (pthread_cond_t *cond, const pthread_condattr_t *attr)
        - Inicializar la variable condición.
    - int pthread_cond_destroy (pthread_cond_t *cond)

Ejemplo:

```
pthread_cond_t vacio, lleno;
pthread_mutex_t mutex;
int hayElemento;
tipo_elemento Buffer;
...
main()
   { ...
     hayElemento= 0;
     pthread_init();
     pthread_cond_init(&vacio, NULL);
     pthread_cond_init(&lleno, NULL);
     pthread_mutex_init(&mutex, NULL);
     ...
   }
```

```
void *productor(void *datos)
   { tipo_element elem;

     while (true)
       { generar_elemento(elem);
         pthread_mutex_lock (&mutex);
         while (hayElemento = = 1)
               pthread_cond_wait (&vacio, &mutex);
          Buffer = elem;
          hayElemento = 1;
          pthread_cond_signal (&lleno);
          pthread_mutex_unlock (&mutex);
       }
  }
```

```
void *consumidor(void *datos)
   {  tipo_element elem;

     while (true))
      { pthread_mutex_lock (&mutex);
        while (hayElemento = = 0)
                pthread_cond_wait (&lleno, &mutex);
        elem= Buffer;
        hayElemento = 0;
        pthread_cond_signal (&vacio);
        pthread_mutex_unlock (&mutex);
        procesar_elemento(elem);
      }
  }
```

Attribute object

- API Pthreads permite que se pueda cambiar los atributos por defecto de las entidades, utilizando attributes objects.
- es una estructura de datos que describe las propiedades de la entidad en cuestión (thread, mutex, variable de condición).
- Una vez que estas propiedades están establecidas, el attribute object es pasado al método que inicializa la entidad.
- Ventajas
    - Esta posibilidad mejora la modularidad.
    - Facilidad de modificación del código.

Semáforos y monitores

- Los threads pueden sincronizar por semáforos (librería semaphore.h).
    - sem_t semaforo→ se declaran globales a los threads.
    - sem_init (&semaforo, alcance, inicial) → en esta operación se inicializa el semáforo semaforo. Inicial es el valor con que se inicializa el semáforo. Alcance indica si es compartido por los hilos de un único proceso (0) o por los de todos los procesos (≠ 0).

    - sem_wait(&semaforo) → equivale al P.
    - sem_post(&semaforo) → equivale al V.
    - Hay funciones extra
- No posee “Monitores” pero con la exclusión mutua y la sincronización por condición se puede simular
    - El acceso exclusivo al monitor se simula usando una variable mutex la cual se bloquea antes de la llamada al procedure y se desbloquea al terminar el mismo (una variable mutex diferente para cada monitor).
    - Cada llamado de un proceso a un procedure de un monitor debe ser reemplazado por el código de ese procedure.

Librerías para manejo de PM

- Diferentes protocolos para Send y Receive.
- Bloqueantes: Bloquea al enviador hasta que el valor este en un lugar seguro (ociosidad en el enviante). Dos posibilidades:
    - Send/Receive bloqueantes sin buffering: similar a PMS
    - Send/Receive bloqueantes con buffering: similar a PMA, hay un buffer en algún lugar de la memoria, seguramente en el receptor. Reduce tiempos ociosos pero usa mas memoria y a veces se hacen copias innecesarias.
- No bloqueantes: No se espera a que el dato este seguro. Puede enviarse algo que en realidad no se quería enviar. Dos posibilidades:
    - Send/Receive no bloqueantes sin buffering: la comunicación se hace cuando el otro hace receive, estohace que el dato este mas tiempo inseguro.
    - Send/Receive no bloqueantes con buffering: inseguro mientras no se copie en buffer (menos tiempo que sin buffering)

MPI

- MPI define una librería estándar que puede ser empleada desde C y otros, tiene un modelo SPMD
- Todas las rutinas, tipos de datos y constantes tienen prefijo MPI.
- Estándar define la sintaxis y la semántica de más de 125 rutinas, pero con 6 ya basta para escribir programas paralelos basados en el PM
- Funciones
    - MPI_Init: se invoca en todos los procesos antes que cualquier otro llamado a rutinas MPI. Sirve para inicializar el entorno MPI. Se suelen pasar los parámetros del main para poder usarlos en cada proceso.
      MPI_Init (int *argc, char **argv)
    - MPI_Finalize: se invoca en todos los procesos como último llamado a rutinas MPI. Sirve para cerrar el entorno MPI.
      MPI_Finalize ()
- Hace uso de comunicadores (variables del tipo MPI_Comm), estos definen el dominio de comunicación, cada proceso puede pertenecer a muchos comunicadores. Hay un comunicador que incluye a todos los procesos de la aplicación (MPI_COMM_WORLD). En cada operación de transferencia se debe indicar el comunicador sobre el que se va a realizar.

- Se puede adquirir información con:
    - MPI_Comm_size: indica la cantidad de procesos en el comunicador. MPI_Comm_size (MPI_Comm comunicador, int *cantidad).
    - MPI_Comm_rank: indica el “rank” (identificador) del proceso dentro de ese comunicador. Cada proceso puede tener un rank diferente en cada comunicador.
      MPI_Comm_rank (MPI_Comm comunicador, int *rank)
- Tipos de datos

Tipo de Datos MPI	Tipo de Datos C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Packed es como un registro puedo enviar varios tipos de datos.

- Diferentes protocolos para Send.
    - Send bloqueantes con buffering (Bsend).
    - Send bloqueantes sin buffering (Ssend).
    - Send no bloqueantes (Isend).
- Diferentes protocolos para Recv.
    - Recv bloqueantes (Recv).
    - Recv no bloqueantes (Irecv).
- Comunicación punto a punto BLOQUEANTE
    - MPI_Send, MPI_Ssend, MPI_Bsend: rutina básica para enviar datos a otro proceso.
        - MPI_Send (void *buf, int cantidad, MPI_Datatype tipoDato, int destino, int tag, MPI_Comm comunicador)
            - 1er parámetro es desde donde envio dato en la memoria
            - 2do parámetro es la cantidad de elementos en el mensaje
            - 3er parámetro es el tipo
            - 4to es el destino
            - 5to etiqueta para identificar el mensaje (es un nro)
            - 6to el comunicador
    - MPI_Recv: rutina básica para recibir datos a otro proceso.

        - MPI_Recv (void *buf, int cantidad, MPI_Datatype tipoDato, int origen, int tag, MPI_Comm comunicador, MPI_Status *estado)
            - 1er parámetro es donde dejo el dato en la memoria
            - 2do parámetro es la cantidad máxima de elementos que me pueden enviar
            - 3er parámetro es el tipo
            - 4to es el origen
            - 5to etiqueta para identificar el mensaje (es un nro)
            - 6to el comunicador
            - 7mo el estado, este contiene el rank y el tag si es que hay comodines
                - Estos comodines pueden estar en el origen y la etiqueta: MPI_ANY_SOURCE y MPI_ANY_TAG._
            - MPI_Get_count para obtener la cantidad de elementos recibidos
              MPI_Get_count(MPI_Status *estado, MPI_Datatype tipoDato, int *cantidad)
- Comunicación punto a punto NO BLOQUEANTE
    - Comienzan la operación de comunicación e inmediatamente devuelven el control (no se asegura que la comunicación finalice correctamente). MPI_Isend (void *buf, int cantidad, MPI_Datatype tipoDato, int destino, int tag, MPI_Comm comunicador, MPI_Request *solicitud)
      MPI_Irecv (void *buf, int cantidad, MPI_Datatype tipoDato, int origen, int tag, MPI_Comm comunicador, MPI_Request *solicitud)
      El request tiene si se termino o no la comunicación, para eso lo debo pasar en los siguientes llamados.
    - MPI_Test: testea si la operación de comunicación finalizó
      MPI_Test (MPI_Request *solicitud, int *flag, MPI_Status *estado)
    - MPI_Wait: bloquea al proceso hasta que finaliza la operación. No tiene sentido ponerlo luego de un send o receive, para eso uso uno bloqueante y listo, solo tiene sentido para adelantar trabajo (para eso ponerlo en el medio)
      MPI_Wait (MPI_Request *solicitud, MPI_Status *estado)
    - Este tipo de comunicación permite solapar cómputo con comunicación. Evita overhead de manejo de buffer y deja en manos del programador asegurar que se realice la comunicación correctamente.
    - Para saber si hay mensajes pendientes con ciertas características uso Probe o Iprobe.

Comunicaciones colectivas

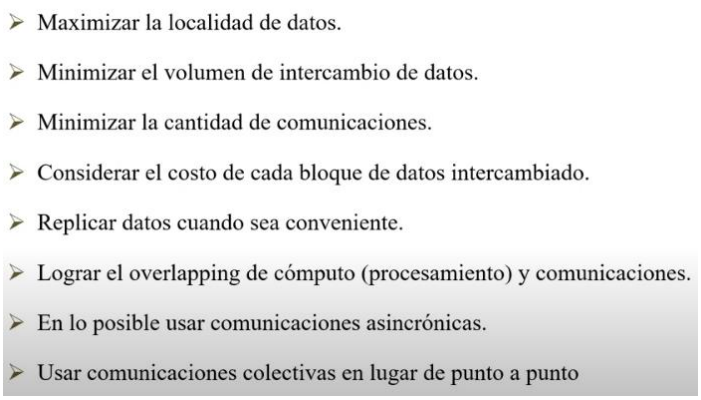
- MPI tiene un conjunto de funciones para realizar operaciones colectivas, sobre un grupo de procesos asociado a un comunicador. Todos los procesos del comunicador deben llamar a la rutina colectiva:

- MPI_Barrier
- MPI_Bcast
- MPI_Scatter - MPI_Scatterv
- MPI_Gather - MPI_Gatherv
- MPI_Reduce
- Otras…

- Este tipo de conexión es super eficiente a comparación de otras librerías.
- Tiene para reducir (reduce) todos los mensajes a uno combinando los elementos enviados por cada uno de los procesos aplicando una cierta operación.
- Existe la operación gather que recolecta el vector de datos de todos los procesos y los concatena en orden para dejar el resultado en un único proceso.
    - Por ejemplo, 4 procesos envían un mensaje cada uno, el gather guarda los cuatro mensajes en uno solo (por ejemplo el 0 guarda los 4 mensajes)
- Scatter es lo opuesto al gather, reparte un vector de datos entre todos los procesos inclusive el mismo dueño del vector. El normal los divide de forma equitativa, el scatterV los divide con una cantidad distinta a cada uno.
    - Si tengo un proceso con 4 mensajes el scatter manda 1 mensaje a cada uno por ejemplo.

Minimizando overheads de comunicación

- Algunos tips:



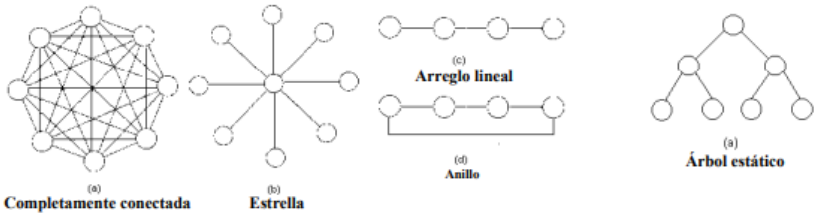
Arquitecturas Paralelas

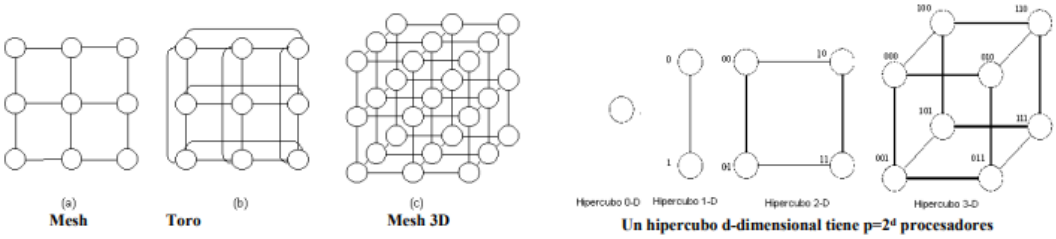
Se clasifican por:

- Por la organización del espacio de direcciones (esto lo tengo en el segundo resumen)
- Por la granularidad (esto esta en el primero, pero acoto con: mayor potencia máquinas, comunicación lenta es grano grueso; menor potencia muchas maquinas, red de comunicación rápida es grano fino)
- Por el mecanismo de control: se basa en la manera en que las instrucciones son ejecutadas sobre los datos.

- SISD (Single Instruction Single Data).
  - La arquitectura va a ejecutar una única instrucción por ciclo de reloj y lo va a hacer sobre un dato particular. Un ejemplo claro son los monoprocesadores que hacían esto. Su ejecución es determinística y secuencial
- SIMD (Single Instruction Multiple Data).
  - La arquitectura va a ejecutar una única instrucción por ciclo de reloj y lo va a hacer sobre un dato particular. Un ejemplo claro son los monoprocesadores que hacían esto. Su ejecución es determinística y secuencial
- MISD (Multiple Instruction Single Data).
  - Una sola instrucción que se aplica a múltiples datos de forma sincrónica y determinística. Hay varios procesadores idénticos con sus memorias donde a estos procesadores se le envía una única instrucción para que ejecuten esa misma instrucción sobre algún dato en su memoria particular.
  - Un ejemplo puede ser un if con una condición para la cual unos procesadores deben ejecutar y otros no, con esto podemos deshabilitar y habilitar de forma selectiva los procesadores para que ejecuten o no.
  - Esta clasificación es adecuada para aplicaciones con un alto grado de regularidad. Se está tomando en cuenta por las GPUs que usan este mecanismo de control.
- MIMD (Multiple Instruction Multiple Data).
  - Es la opción usada hoy en día para hacer paralelismo, es de propósito general y es la arquitectura sobre la cual se ejecutan las aplicaciones vistas en la materia. Cada unidad de procesamiento ejecuta su propia instrucción sobre sus propios datos , de forma asincrónica, a su propio ritmo. Pueden tener memoria compartida o memoria distribuida. Ejemplo son los clusters o los clusters de multicore.
  - Tienen dos subclasificaciones
    - **MPMD (multiple program multiple data):** Cada procesador ejecuta su propio programa
    - **SPMD (single program multiple data):** Hay un único programa fuente y cada procesador ejecutara su copia independiente.

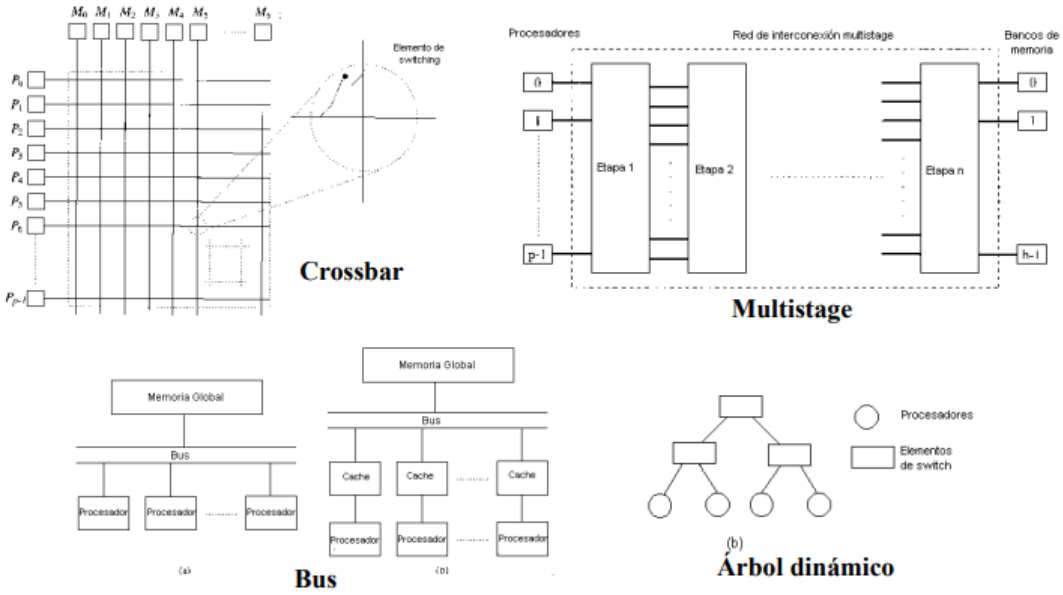
- Por red de interconexión
  - Redes de interconexión estáticas**:** Son links punto a punto para conectar los módulos, usualmente se utiliza en máquinas de pasaje de mensajes. En ellas cada nodo es una computadora completa.
  - Ejemplos:





- Completamente conectada es la más rápida de todas ya que hay comunicación directa en un solo paso (hay link directo). Un nuevo procesador agregado implica agregar un montón de links.
- La conexión en estrella es un nodo central y todos los demás se conectan a él, agregar un nuevo nodo es agregar un solo link, lo malo es que la comunicación entre procesos que no sean nodos central, entonces la comunicación es en 2 pasos.
- El arreglo lineal o anillo, es una secuencia de nodos, cada uno conoce a su vecino izquierdo y derecho y los extremos conocen un solo vecino, la comunicación es lenta, si quiero conectar los procesos del extremo debo comunicarme en 3 pasos. En el caso del anillo se reduce la comunicación de extremos ya que estos están unidos, el peor caso es de un extremo al medio.
- En el árbol estático, las aristas son las conexiones, se reduce la cantidad de pasos.
- Los hipercubos, permiten asegurar que la comunicación se va a hacer como máximo en D pasos.

- Redes de interconexión dinámica: Son construidas usando switches y enlaces de comunicación, este caso se piensa para máquinas de memoria compartida.
- Ejemplos:



- En crossbar se activa un switch por cada bloque de memoria, es la red más rápida de todas pero lo malo que tiene es que es costosa de implementar y agregar procesador/memoria es agregar nuevos cables.

- En buses hay un único canal a través del cual las unidades de procesamiento se comunican y acceden a memoria, es la menos costosa si se quiere agregar un CPU o más memoria pero es la más lenta para las comunicaciones o memoria.

- En multistage se pasa por una serie de etapas que cada una tiene un conjunto de switch que van desviando las CPUs a los bloques de memoria deseado, es rápido y es fácil de ampliar. Es más un caso intermedio.

Diseño de Algoritmos Paralelos

Hay algunas soluciones secuenciales que son optimas pero pueden ser poco paralelizables y otras que no son muy optimas pero si muy paralelizables. En el medio de una paralelización se puede encontrar mejores soluciones y también se deben tener en cuenta las optimizaciones haciendo que sea difícil la programación paralela.

Los aspectos independientes de la máquina se consideran de forma temprana mientras que los aspectos específicos se demoran. Esto se termina simplificándose en 2 pasos fundamentales: Primero la descomposición del problema en tareas concurrentes (no toma en cuenta la arquitectura) y el mapeo de las tareas en las unidades de procesamiento.

Para esto se toman en cuenta:

- Cuál es la granularidad de las tareas, que conviene respecto a la arquitectura
- De qué forma se mapean las tareas y datos a las distintas unidades de procesamiento

- Como se manejará la comunicación y sincronización entre procesos. Si es una arquitectura distribuida no hay memoria compartida, si hay memoria compartida se debe elegir como hacer la comunicación.
- Asegurar la corrección del programa (asegurar resultados correctos). Evitar deadlocks y desbalances (un procesador trabaja más que otro).
- Tener un cierto grado de tolerancia a fallo.
- Manejar la heterogeneidad.
- Lograr que el sistema sea escalable.
- Consumo energético.

Para diseñar algoritmo paralelo se debe tener en cuenta algunos de los siguientes puntos:

- Identificar tareas concurrentes
- Mapear tareas a procesos en distintos procesadores.
- Distribuir datos de entrada, intermedios y de salida.
- Manejo de acceso a datos compartidos.
- Sincronizar procesos.

Descomposición

Se descompone el problema en componentes funcionales de forma concurrente. Al principio se plantean tareas lo más pequeñas posibles, teniendo una descomposición de grano fino para obtener la máxima concurrencia posible.

En etapas posteriores se analiza cada tarea, su comunicación, sincronización y la arquitectura a usar y se juntan distintas tareas para obtener una composición de grano grueso. Estas tareas de grano grueso se van a poder convertir en procesos o hilos para después mapearlos.

Dos tipos:

- Descomponer el dominio (paralelismo de datos): Se descomponen los datos, cada tarea de igual código trabaja sobre conjuntos diferentes de esos datos.

- Descomponer funcionalmente (paralelismo funciona): Primero se descompone el cómputo en tareas distintas (en distintas funciones)y luego se distribuyen los datos. Los datos pueden ser necesitados por una tarea o entre varios, teniendo que revisar cuando comunicar la información entre las tareas. Es muy usada para darle estructura a un programa, para dar más claridad.

- Se puede tener una descomposición híbrida, primero una descomposición funcional y luego una descomposición de datos sobre las tareas resultantes.

Nota: Se puede hacer mayor descomposición de datos que funcional. La descomposición de datos tiene mas granularidad.

Aglomeración

Se considera si es útil combinar o aglomerar las tareas para obtener otras de mayor tamaño y si vale la pena replicar datos y/o computación.

Hay 3 objetivos a seguir en la aglomeración y replicación:

- Incremento de la granularidad de tareas: Juntar tareas que tengan mucha comunicación entre ellas para reducir la cantidad de comunicaciones. No conviene juntar tareas que se pueden ejecutar de forma concurrente.
- Preservación de la flexibilidad: Al juntar tareas se puede limitar la escalabilidad del algoritmo, la idea es lograr una composición de grano grueso sin sacrificar tanta portabilidad y escalabilidad del algoritmo.
- Reducción de costos de IS: Se intenta evitar cambios extensivos.

Una vez finalizada la descomposición se tienen un conjunto de tareas que tienen ciertas características que pueden ser útiles para la etapa de mapeo: La generación de las tareas, su tamaño (si son todas del mismo o no), el conocimiento del tamaño de las tareas (si se que no son del mismo tamaño, se puede predecir cuanto tardaran o no con diferentes datos) y el volumen de datos asociado a cada tarea.

Mapeo

Se distribuyen las tareas en las distintas unidades de procesamiento, no sucede en uniprocesadores o en máquinas de memoria compartida con scheduling automático

Distribuir tareas en unidades de procesamiento → Mapeo

Distribuir tareas a procesos y hay un unico proceso en cada unidad → Scheduling

- Objetivo: Minimizar TE de app completa
- Dos estrategias que a veces conflictúan: La primera es ubicar tareas que pueden ejecutar concurrentemente en diferentes procesadores para mejorar la concurrencia o poner tareas que se comunican con frecuencia en procesadores iguales para incrementar la localidad y la sobrecarga de comunicación.

Normalmente hay mas tareas que procesadores físicos, el lenguaje de especificación de algoritmos paralelos debe poder indicar las tareas que pueden ejecutarse concurrentemente y su prioridad para el caso que no haya suficientes procesadores para atenderlas.

Lograr un buen mapping es crítico para el rendimiento de algoritmos paralelos

Hay que:

1. Tratar de mapear tareas independientes a distintos procesadores

2. Asignar prioritariamente los procesadores disponibles a tareas que están en un camino crítico

3. Asignar las tareas con alto nivel de interacción al mismo procesador, de modo de disminuir el tiempo de comunicación físico.

Métricas de rendimiento

En un algoritmo paralelo es interesante saber cuál es la ganancia en performance. El tiempo de ejecución depende del tamaño de la entrada y de la arquitectura y número

de procesadores. Es complejo analizar la performance, depende de lo que se quiere medir, de qué sistema es mejor que otro y qué sucede si se agregan procesadores.

Speedup

Sirve para saber cuanto mejor es la solución paralela respecto a la secuencial. Nos permite independizarnos del tamaño del problema (pero no de la arquitectura)

Tiempos a tener en cuenta:

- Tp → tiempo desde que empieza a ejecutarse el primer hilo hasta que termina el ultimo.
- Ts (Taylor Swift te amo) → tiempo que tarda en ejecutarse la mejor solución secuencial sobre la mejor maquina.
- S → Speedup → Ts/Tp.
- S optimo → Depende de la arquitectura, si es homogénea es la cantidad de procesadores, si es heterogénea es la siguiente formula

$$S_{óptimo} = \sum_{i=0}^{P} \frac{PotenciaCalculo(i)}{PotenciaCalculo(mejor)}$$

- Rango de valores: en general entre 0 y S óptimo. Si es > a 1 es mejor el paralelo.
- El speedup puede ser lineal o perfecto (igual al optimo), sublínea (menor optimo) y superlineal (mayor optimo). Si es igual al optimo significa que estamos usando la arquitectura de forma perfecta.

Eficiencia

Sirve para saber que tan cerca estamos del Speedup óptimo de nuestra arquitectura. Nos permite independizarnos de la arquitectura. Básicamente me permite decir que tan bien estoy aprovechando la arquitectura.

- E = S/S optimo
- Valores entre 0 y 1. Cuando es 1 corresponde al speedup perfecto. Nunca se dará ya que hay ciertos factores que lo limitan como:
    - Alto porcentaje de código secuencial (Ley de Amdahl).
    - Alto porcentaje de entrada/salida respecto de la computación.
    - Algoritmo no adecuado (necesidad de rediseñar).
    - Excesiva contención de memoria (rediseñar código para localidad de datos).
    - Tamaño del problema (puede ser chico, o fijo y no crecer con p).
    - Desbalance de carga (produciendo esperas ociosas en algunos procesadores).
    - Overhead paralelo: ciclos adicionales de CPU para crear procesos, sincronizar, etc

Escalabilidad

Hay que tratar de que la solución sea escalable, existen ciertas métricas que me permiten analizarla.

Paradigmas de Programación Paralela

Un paradigma de programación son clases de algoritmos que resuelven distintos problemas pero tienen la misma estructura de control y además similares patrones de comunicación y sincronización.

Cliente/Servidor

- Predominante en las aplicaciones de procesamiento distribuido
- Servidores procesos que esperan pedidos de servicios de múltiples clientes.
- Interacción bidireccional.
- Lo que cambia en el paradigma es lo que el cliente hace y la forma en que van a interactuar entre ellos.
- Varios mecanismos de invocación pero los que más suelen servir son aquellos con comunicación bidireccional como RPC, Rendezvous o incluso Monitores. Con pasaje de mensajes se puede implementar pero se necesita un canal para la comunicación y otro para la respuesta. Aparte que habría uno de pedido por cada cliente y uno de respuesta por cada uno.

Master/slave o master/worker

El master envía iterativamente datos a los workers y recibe resultados de estos. Hay un posible cuello de botella por ejemplo por tareas muy chicas o slaves muy rápidos.

Si el master solo enviará información no sería necesario en memoria compartida porque la info a enviar podría ser compartida. Normalmente se usa en memoria distribuida.

Dos casos de acuerdo con las dependencias de las iteraciones:

- Iteraciones dependientes: El master necesita los resultados de todos los workers para generar un nuevo conjunto de datos.
- Entradas de datos independientes: Los datos llegan al maestro, quien no necesita resultados anteriores para generar un nuevo conjunto de datos.

También hay dos opciones para la distribución de datos:

- Distribuir todos los disponibles de acuerdo a alguna política (estática).
- Bajo petición o demanda (dinámico).
- Puede haber un hibrido donde al inicio da muchos datos (pero no todos) y luego reparte bajo demanda.

Pueden darse distintos casos:

- Procesadores heterogéneos y con distintas velocidades: Esto puede dar problemas con el balance de carga.
- Trabajo que debe realizarse en fases, esto requiere sincronización
- Generalización a modelo multi nivel o jerárquico.

Pipeline y Algoritmos Sistólicos

El problema se particiona en una secuencia de pasos. El stream de datos pasa entre los procesos y cada uno realiza una tarea sobre él.

Un ejemplo es el filtrado, etiquetado y análisis de escena en imágenes.

Es un mapeo natural a un arreglo lineal de procesadores.

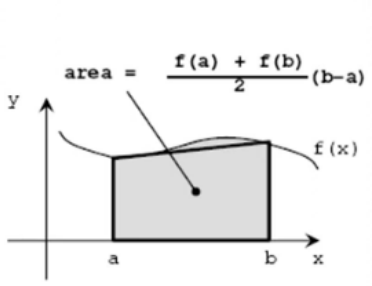
Dividir y Conquistar

Implica paralelismo recursivo en donde el problema general puede descomponerse en procesos recursivos que trabajan sobre partes del conjunto total de datos. Sale de un programa secuencial que es recursivo. En lugar de hacer llamados recursivos en el programa principal lo que va a hacer es generar nuevos procesos que trabajan con datos más chicos que van a estar ejecutándose.

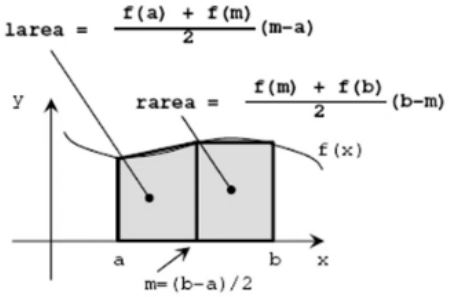
División repetida de problemas y datos en subproblemas más chicos (fase de dividir); Una vez que ya se puede resolver el subproblema generado comienza la etapa de conquistar donde se resuelven estos problemas de forma independiente, usualmente de manera recursiva. Las soluciones son combinadas en la solución global.

La subdivisión puede corresponderse con la descomposición entre procesadores.

Procedimiento recursivo adaptivo



(a) First approximation (area)



(b) Second approximation (larea + rarea)

Si ***abs((larea + rarea) - area) > e***, repetir el cómputo para cada intervalo ***[a,m]*** y ***[m,b]*** de manera similar hasta que la diferencia entre aproximaciones consecutivas esté dentro de un dado ***e.***

Procedimiento secuencial

```
double quad(double l,r,fl,fr,area) {
  double m = (l+r)/2;
  double fm = f(m);
  double larea = (fl+fm)*(m-l)/2;
  double rarea = (fm+fr)*(r-m)/2;
  if (abs((larea+rarea)-area) > e) {
     larea = quad(l,m,fl,fm,larea);
     rarea = quad(m,r,fm,fr,rarea);

  }
  return (larea+rarea);
}
```

Procedimiento paralelo

```
double quad(double l,r,fl,fr,area) {
   double m = (l+r)/2;
   double fm = f(m);
   double larea = (fl+fm)*(m-l)/2;
   double rarea = (fm+fr)*(r-m)/2;
   if (abs((larea+rarea)-area) > e) {
      co larea = quad(l,m,fl,fm,larea);
      || rarea = quad(m,r,fm,fr,rarea);
      oc
   }
   return (larea+rarea);
}
```

- Dos llamados recursivos son independientes y pueden ejecutarse en paralelo.
- Uso: area = quad (a, b ,f(a), f(b), (f(a) + f(b)) * (b-a) /2)

SPMD

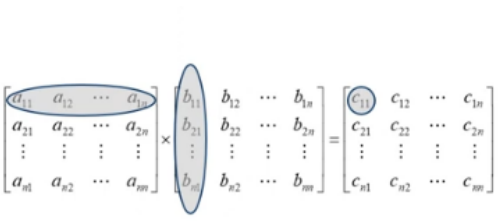
Van a haber varios procesos que todos van a ejecutar el mismo proceso pero con distintos sectores de datos. La diferente evaluación de un predicado en sentencias condicionales permite que cada nodo toma distintos caminos del programa.

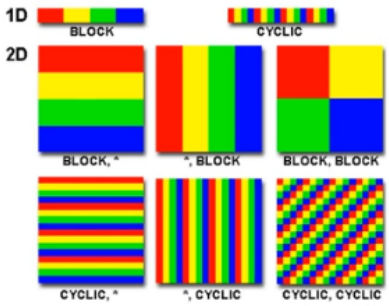
Tiene 2 fases: La elección de la distribución de datos y la generación del programa paralelo.

La primera determina el lugar que ocuparan los datos en los nodos.

La segunda convierte al programa secuencial en SPMD. En la mayoría de los lenguajes depende de la distribución de datos.

Suele implicar un paralelismo iterativo, donde un programa consta de un conjunto de procesos los cuales tienen 1 o más loops. Cada proceso es un programa iterativo. Generalmente el dominio de datos se divide entre los procesos siguiendo diferentes patrones.



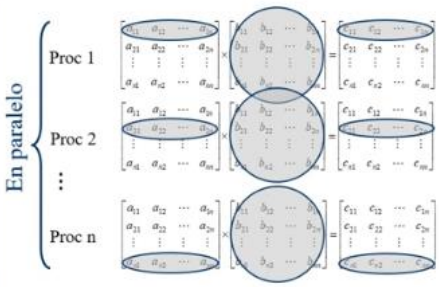


Ejemplo con multiplicación de matrices: Son 3 for anidados, no voy a entrar en detalle.

Paralelizado cada elemento resultado de la matriz se podría estar ejecutando independientemente. Lo que hay que tener en cuenta con cuántos procesos contamos, si usáramos N^2 procesos (con una matriz de tamaño n) y hay procesadores más rápidos que otros, esto haría que se desperdicie tiempo, porque unos terminaron mientras que los otros no.

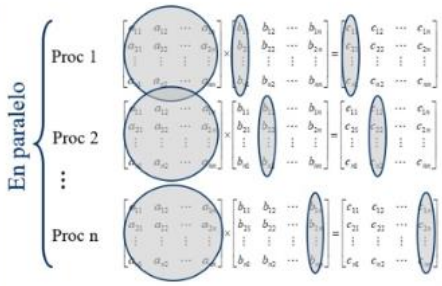
Solución paralela por fila:

```
double a[n,n], b[n,n], c[n,n];
co [i = 1 to n]
  { for [j = 1 to n]
     { c[i,j] = 0;
       for [k = 1 to n]
         c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
     }
  }
```



Solución paralela por columna:

```
double a[n,n], b[n,n], c[n,n];
co [j = 1 to n]
  { for [i = 1 to n]
     { c[i,j] = 0;
       for [k = 1 to n]
         c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
     }
  }
```



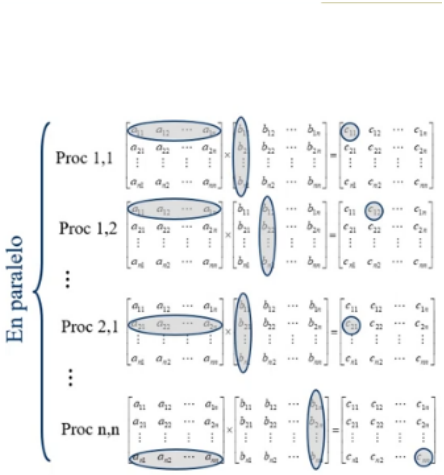
En la primera cada proceso calcula una fila de la matriz resultado y en la segunda una columna de la matriz resultante.

Solución paralela por celda (opción 1):

```
double a[n,n], b[n,n], c[n,n];
co [i = 1 to n , j= 1 to n]
     { c[i,j] = 0;
       for [k = 1 to n]
         c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
     }
```

Solución paralela por celda (opción 2):

```
double a[n,n], b[n,n], c[n,n];
co [i = 1 to n]
  { co [j = 1 to n]
     { c[i,j] = 0;
       for [k = 1 to n]
         c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
     }
  }
```



En la primera es un co que genera N filas, en la segunda a su vez crea n hilos por la cantidad de columnas.

El problema es que no tenemos n procesadores, por lo que tenemos que hacer que cada proceso ejecute un grupo de filas (solución de grano un poco más grueso).

El grupo óptimo de filas a ejecutar es un problema interesante para balancear el costo de procesamiento con costo de comunicaciones.