Programación Concurrente ATIC
Redictado Programación Concurrente

Clase 10

Facultad de Informática

UNLP

Paradigmas de Interacción entre Procesos

Paradigmas para la interacción entre procesos

- 3 esquemas básicos de interacción entre procesos: *productor/consumidor*, *cliente/servidor* e *interacción entre pares*.
- Estos esquemas básicos se pueden combinar de muchas maneras, dando lugar a otros **paradigmas** o modelos de interacción entre procesos.

Paradigma 1: *master / worker*
Implementación distribuida del modelo *Bag of Task*.

Paradigma 2: *algoritmos heartbeat*
Los procesos periódicamente deben intercambiar información con mecanismos tipo send/receive.

Paradigma 3: *algoritmos pipeline*
La información recorre una serie de procesos utilizando alguna forma de receive/send.

Paradigmas para la interacción entre procesos

Paradigma 4: *probes (send) y echoes(receive)*
La interacción entre los procesos permite recorrer grafos o árboles (o estructuras dinámicas) diseminando y juntando información.

Paradigma 5: *algoritmos broadcast*
Permiten alcanzar una información global en una arquitectura distribuida. Sirven para toma de decisiones descentralizadas.

Paradigma 6: *token passing*
En muchos casos la arquitectura distribuida recibe una información global a través del viaje de tokens de control o datos. También permite la toma de decisiones distribuidas.

Paradigma 7: *servidores replicados*
Los servidores manejan (mediante múltiples instancias) recursos compartidos tales como dispositivos o archivos.

Paradigmas para la interacción entre procesos

Manager/Worker

- El concepto de ***bag of tasks*** usando variables compartidas supone que un conjunto de workers comparten una "bolsa" con tareas independientes. Los workers sacan una tarea de la bolsa, la ejecutan, y posiblemente crean nuevas tareas que ponen en la bolsa (ejemplo en LINDA manejando un espacio compartido de tuplas).

- La mayor virtud de este enfoque es la escalabilidad y la facilidad para equilibrar la carga de trabajo de los workers.

- Analizaremos la implementación de este paradigma con mensajes en lugar de MC. Para esto un proceso *manager* implementará la "bolsa" manejando las tasks, comunicándose con los workers y detectando fin de tareas. **Se trata de un esquema C/S**.

- Ejemplo: multiplicación de matrices ralas.

Paradigmas para la interacción entre procesos

Heartbeat

- Paradigma *heartbeat* ⇒ útil para soluciones iterativas que se quieren paralelizar.
- Usando un esquema "*divide & conquer*" se distribuye la carga (datos) entre los workers; cada uno es responsable de actualizar una parte.
- Los nuevos valores dependen de los mantenidos por los workers o sus vecinos inmediatos.
- Cada "paso" debiera significar un progreso hacia la solución.
- Formato general de los worker:

```
process worker [i =1 to numWorkers]
  {  declaraciones e inicializaciones locales;
     while (no terminado)
       {  send valores a los workers vecinos;
          receive valores de los workers vecinos;
          Actualizar valores locales;
       }
  }
```

- Ejemplo: grid computations (imágenes), autómatas celulares (simulación de fenómenos como incendios o crecimiento biológico).

Paradigmas para la interacción entre procesos

Heartbeat - Topología de una red

Los procesadores están conectados por canales bidireccionales. Cada uno se comunica sólo con sus vecinos y conoce esos links.

¿Cómo puede cada procesador determinar la topología completa de la red?

- Modelización:
  - Procesador ⇒ proceso
  - Links de comunicación ⇒ canales compartidos.
- Soluciones: los vecinos interactúan para intercambiar información local.

Algoritmo Heartbeat: se expande enviando información; luego se contrae incorporando nueva información.

- Procesos ***Nodo[p:1..n]***.
- Vecinos de ***p***: ***vecinos[1:n]*** → vecinos[q] es true si q es vecino de p.
- ***Problema:*** computar ***top*** (matriz de adyacencia), donde ***top[p,q]*** es true si p y q son vecinos.

Paradigmas para la interacción entre procesos

Heartbeat - Topología de una red

Cada nodo debe ejecutar un nº de rondas para conocer la topología completa. Si el diámetro ***D*** de la red es conocido se resuelve con el siguiente algoritmo.

```
chan topologia[1:n] ([1:n,1:n] bool)

Process Nodo[p:1..n]
{  bool vecinos[1:n],  bool nuevatop[1:n,1:n],  top[1:n,1:n] = ([n*n] false);
   top[p,1..n] = vecinos;
   int r = 0;

   for (r =0 ; r < D; r++)
      { for [q = 1 to n st vecinos[q] ] send topologia[q](top);
        for [q = 1 to n st vecinos[q] ]
          {  receive topologia[p](nuevatop);
             top = top or nuevatop;
          }
        r = r + 1;
      }
 }
```

Paradigmas para la interacción entre procesos

Heartbeat - Topología de una red

- Rara vez se conoce el valor de ***D***.
- Excesivo intercambio de mensajes ⇒ los procesos cercanos al "centro" conocen la topología más pronto y no aprenden nada nuevo en los intercambios.
- El tema de la terminación ⇒ ¿local o distribuida?
- ***¿Cómo se pueden solucionar estos problemas?***
  - Después de ***r*** rondas, ***p*** conoce la topología a distancia ***r*** de él. Para cada nodo ***q*** dentro de la distancia ***r*** de ***p***, los vecinos de ***q*** estarán almacenados en la fila ***q*** de ***top*** ⇒ ***p*** ejecutó las rondas suficientes tan pronto como cada fila de ***top*** tiene algún valor ***true***.
  - Luego necesita ejecutar una última ronda para intercambiar la topología con sus vecinos.
- No siempre la terminación se puede determinar localmente.

Paradigmas para la interacción entre procesos

Heartbeat - Topología de una red

```
chan topologia[1:n](emisor : int; listo : bool; top : [1:n,1:n] bool)

Process Nodo[p:1..n]
{  bool vecinos[1:n], activo[1:n] = vecinos, top[1:n,1:n] = ([n*n]false), nuevatop[1:n,1:n];
   bool qlisto, listo = false;
   int r = 0, int emisor;
   top[p,1..n] = vecinos;
   while (not listo)
     {   for [q = 1 to n st activo[q] ] send topologia[q](p,false,top);
         for [q = 1 to n st activo[q] ]
            {   receive topologia[p](emisor,qlisto,nuevatop);
                top = top or nuevatop;
                if (qlisto) activo[emisor] = false;
             }
          if (todas las filas de top tiene 1 entry true) listo=true;
          r := r + 1;
      }
    for [q = 1 to n st activo[q] ] send topologia[q](p,listo,top);
    for [q=1 to n st activo[q]] receive topologia[p](emisor,d,nuevatop);
}
```

Paradigmas para la interacción entre procesos
Pipeline

- Un pipeline es un arreglo lineal de procesos “filtro” que reciben datos de un puerto (canal) de entrada y entregan resultados por un canal de salida.
- Estos procesos (“workers”) pueden estar en procesadores que operan en paralelo, en un primer esquema *a lazo abierto* (W_1 en el INPUT, W_n en el OUTPUT).
- Un segundo esquema es el pipeline *circular*, donde W_n se conecta con W_1. Estos esquemas sirven en procesos iterativos o bien donde la aplicación no se resuelve en una pasada por el pipe.
- En un tercer esquema posible (*cerrado*), existe un proceso coordinador que maneja la “realimentación” entre W_n y W_1.
- Ejemplo: multiplicación de matrices en bloques.

Paradigmas para la interacción entre procesos

Probe-Echo

- Arboles y grafos son utilizados en muchas aplicaciones distribuidas como búsquedas en la WEB, BD, sistemas expertos y juegos.
- Las arquitecturas distribuidas se pueden asimilar a los nodos de grafos y árboles, con canales de comunicación que los vinculan.
- DFS es uno de los paradigmas secuenciales clásicos para visitar todos los nodos en un árbol o grafo. Este paradigma es el análogo concurrente de DFS.
- ***Prueba-eco*** se basa en el envío de un mensajes (“probe”) de un nodo al sucesor, y la espera posterior del mensaje de respuesta (“echo”).
- Los **probes** se envían en paralelo a todos los sucesores.
- Los algoritmos de prueba-eco son particularmente interesantes cuando se trata de recorrer redes donde no hay (o no se conoce) un número fijo de nodos activos (ejemplo: redes móviles).

Paradigmas para la interacción entre procesos

Broadcast

- En la mayoría de las LAN cada procesador se conecta directamente con los otros. Estas redes normalmente soportan la primitiva ***broadcast***:

  broadcast ch(m);

- Los mensajes broadcast de un proceso se encolan en los canales en el orden de envío, pero broadcast no es atómico y los mensajes enviados por procesos A y B podrían ser recibidos por otros en distinto orden.

- Se puede usar broadcast para diseminar información o para resolver problemas de sincronización distribuida. Ejemplo: semáforos distribuidos, la base es un ***ordenamiento total de eventos de comunicación*** mediante el uso de ***relojes lógicos***.

Paradigmas para la interacción entre procesos

Token Passing

- Un paradigma de interacción muy usado se basa en un tipo especial de mensaje (“token”) que puede usarse para otorgar un permiso (control) o recoger información global de la arquitectura distribuida. Un ejemplo del primer tipo de algoritmos es el caso de tener que controlar ***exclusión mutua distribuida***.
- Ejemplos de recolección de información de estado son los algoritmos de detección de terminación en computación distribuida.
- Aunque el problema de la SC se da principalmente en programas de MC, puede encontrarse en programas distribuidos cuando hay algún recurso compartido que puede usar un único proceso a la vez. Generalmente es una componente de un problema más grande, tal como asegurar consistencia en un sistema de BD.
- Soluciones posibles: Monitor activo que da permiso de acceso (ej: locks en archivos), semáforos distribuidos (usando broadcast, con gran intercambio de mensajes), o *token ring* (descentralizado y fair).

Paradigmas para la interacción entre procesos

Servidores Replicados

- Un server puede ser replicado cuando hay múltiples instancias de un recurso: cada server maneja una instancia.
- La replicación también puede usarse para darle a los clientes la sensación de un único recurso cuando en realidad hay varios.
- Ejemplo: problema de los filósofos
  - Modelo ***centralizado*****:** los Filósofo se comunican con ***UN*** proceso Mozo que decide el acceso o no a los recursos.
  - Modelo ***distribuido*****:** supone **5 procesos Mozo,** cada uno manejando un tenedor. Un Filósofo puede comunicarse con **2** Mozos (izquierdo y derecho), solicitando y devolviendo el recurso. Los Mozos **NO se comunican entre ellos**.
  - Modelo ***descentralizada*****:** cada Filósofo ve **un único** Mozo. Los Mozos se comunican entre ellos (cada uno con sus **2** vecinos) para decidir el manejo del recurso asociado a "su" Filósofo.

Programación Paralela

Clasificación

- ***Programa Concurrente*:** múltiples procesos.
- ***Programa Distribuido*:** programa concurrente en el cual los procesos se comunican y sincronizan por PM, RPC o Rendezvous.
- ***Programa Paralelo*:** programa concurrente escrito para resolver un problema en menos tiempo que el secuencial. El objetivo principal es reducir el tiempo de ejecución, o resolver problemas más grandes o con mayor precisión en el mismo tiempo.
- Un programa paralelo puede escribirse usando VC o PM. La elección la dicta el tipo de arquitectura.

Computación Científica

- Los dos modos tradicionales del descubrimiento científico son ***teoría*** y ***experimentación***.
- El 3er modo es la ***modelización computacional***, que usa computadoras para simular fenómenos y tratar cuestiones del tipo ***"what if?"***
- Entre las diferentes aplicaciones de cómputo científicas y modelos computacionales existen tres técnicas fundamentales:
  - Computación de grillas (por ejemplo imágenes). Dividen una región espacial en un conjunto de puntos.
  - Computación de partículas (modelos que simulan interacciones de partículas individuales como moléculas u objetos estelares).
  - Computación de matrices (sistemas de ecuaciones simultáneas).

Necesidad del paralelismo

- Ejemplo: ***simulación de circulación oceánica***. División del océano en 4096x1024 regiones, y cada una en 12 niveles. Aproximadamente 50 millones de celdas 3D. Una iteración del modelo simula la circulación por 10 minutos y requiere alrededor de 30 billones de cálculos de punto flotante. Se intenta usar el modelo para simular la circulación en un período de años...

- Problemas "***grand challenge***". Abarcan química cuántica, mecánica estadística, cosmología y astrofísica, dinámica de fluidos computacional, diseño de materiales, biología, farmacología, secuencia genómica, ingeniería genética, medicina y modelización de órganos y huesos humanos, pronóstico del tiempo, sensado remoto, física de partículas etc.

Diseño de algoritmos paralelos

- No se reduce a simples recetas, sino que es necesaria la ***creatividad***. La mejor solución puede diferir totalmente de la sugerida por los algoritmos secuenciales existentes.
- Pero puede darse un enfoque metódico para maximizar el rango de opciones consideradas, brindar mecanismos para evaluar las alternativas, y reducir el costo de ***backtracking*** por malas elecciones ⇒ metodología de diseño que da un enfoque exploratorio en el cual aspectos independientes de la máquina tales como la concurrencia son considerados temprano, y los aspectos específicos de la máquina se demoran.
- 2 etapas:
  - Descomposición.
  - Mapeo

Métricas del paralelismo

- En el mundo serial la performance con frecuencia es medida teniendo en cuenta los requerimientos de tiempo y memoria de un programa.
- En un algoritmo paralelo para resolver un problema interesa saber cuál es la ganancia en performance.
- Hay otras medidas que deben tenerse en cuenta siempre que favorezcan a sistemas con mejor tiempo de ejecución.
- A falta de un modelo unificador de cómputo paralelo, el tiempo de ejecución depende del tamaño de la entrada y de la arquitectura y número de procesadores (*sistema paralelo = algoritmo + arquitectura sobre la que se implementa*).

Métricas del paralelismo

- La diversidad torna complejo el análisis de performance...
  - ¿Qué interesa medir?
  - ¿Qué indica que un sistema paralelo es mejor que otro?
  - ¿Qué sucede si agrego procesadores?
- En la medición de performance es usual elegir un problema y testear el tiempo variando el número de procesadores. Aquí subyacen las nociones de speedup y eficiencia, y la *ley de Amdahl*.
- Otro tema de interés es la *escalabilidad*, que da una medida de usar eficientemente un número creciente de procesadores.

Métricas del paralelismo

Tamaño del Problema (W)

- Función del tamaño de la entrada. Está dado por el número de operaciones básicas necesarias para resolver el problema en el algoritmo secuencial más rápido.
- Es incorrecto pensar, por ejemplo, que en problemas con matrices de *nxn* el tamaño de problema es *n* pues la interpretación cambiaría de un problema a otro. Por ejemplo, duplicar el tamaño de la entrada resulta en un incremento de 8 veces en el tiempo de ejecución serial para la multiplicación y de 4 veces para la suma.
- El ***tiempo de ejecución paralelo***, para un sistema paralelo dado, es función del tamaño del problema y el número de procesadores ($Tp(W,p)$).

Métricas del paralelismo

Speedup (S)

- S es el cociente entre el tiempo de ejecución del algoritmo serial conocido más rápido (T_s) y el tiempo de ejecución paralelo del algoritmo elegido (T_p):

$$S = \frac{T_S}{T_P}$$

- Speedup óptimo depende de la arquitectura (en homogénea P).

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

- Rango de valores: en general entre *0* y $S_{óptimo}$
- Speedup lineal o perfecto, sublineal y superlineal.

Métricas del paralelismo

Eficiencia (E)

- Cociente entre Speedup y Speedup Óptimo.

$$E = \frac{S}{S_{óptimo}}$$

- Mide la fracción de tiempo en que los procesadores son *útiles* para el cómputo.

- El valor está entre 0 y 1, dependiendo de la efectividad de uso de los procesadores. Cuando es 1 corresponde al speedup perfecto.

Métricas del paralelismo

Factores que limitan el Speedup

- Alto porcentaje de código secuencial (*Ley de Amdahl*).
- Alto porcentaje de entrada/salida respecto de la computación.
- Algoritmo no adecuado (necesidad de rediseñar).
- Excesiva contencíón 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.

***Función de overhead*:** $To(W,p) = pTp - W$

Suma todos los overheads en que incurren todos los procesadores debido al paralelismo.

Métricas del paralelismo
Costo

- El costo de un sistema paralelo es el producto de T_p y p.
- Refleja la suma del tiempo que cada procesador utiliza en la resolución del problema.
- Puede expresarse la eficiencia como el cociente entre el tiempo de ejecución del algoritmo secuencial conocido más rápido y el costo de resolver el problema en p procesadores
- También suele referirse como *trabajo*.

Métricas del paralelismo

Grado de concurrencia o paralelismo

- $C(W)$ es el número máximo de tareas que pueden ejecutarse simultáneamente en cualquier momento del algoritmo paralelo.
- Para un W dado, el algoritmo paralelo no puede usar más de $C(W)$ procesadores.
- $C(W)$ depende sólo del algoritmo, no de la arquitectura.
- Supone un número ilimitado de procesadores y otros recursos, lo que no siempre es posible de tener.

Noción de granularidad

- Cuando el número de procesadores crece, normalmente la cantidad de procesamiento en cada una disminuye y las comunicaciones aumentan. Esta relación se conoce como ***granularidad***.
- Puede definirse la granularidad de una aplicación o una máquina paralela como la relación entre la cantidad mínima o promedio de operaciones aritmético-lógicas con respecto a la cantidad mínima o promedio de datos que se comunican.
- La relación cómputo/comunicación impacta en la complejidad de los procesadores: a medida que son más independientes y realizan más operaciones A-L entre comunicaciones, también deben ser más complejos.
- Si la granularidad del algoritmo es diferente a la de la arquitectura, normalmente se tendrá pérdida de rendimiento.

Bibliotecas Actuales

Librería Pthreads

Semáforos con Pthreads

Thread: proceso "liviano" que tiene su propio contador de programa y su pila de ejecución, pero no controla el "contexto pesado" (por ejemplo, las tablas de página).

- Algunos sistemas operativos y lenguajes proveen mecanismos para permitir la programación de aplicaciones "multithreading".
- En principio estos mecanismos fueron heterogéneos y poco portables $\Rightarrow$ a mediados de los 90 la organización POSIX auspició el desarrolló de una biblioteca en C para multithreading **(*Pthreads*).**
- Con esta biblioteca se pueden crear threads, asignarles atributos, darlos por terminados, identificarlos, etc.

Semáforos con Pthreads

 # include <pthread.h>

- Declaración de variables para descriptores de thread:

  pthread_t pid;

- Creación de thread:

  pthread_create(&tid, &attr, start_func, arg);

  - ✓ ***&tid*** es la dirección de un descriptor que se llena si la creación tiene éxito.
  - ✓ ***&attr*** es la dirección de un descriptor inicializado previamente.
  - ✓ el thread comienza la ejecución llamando a ***start_func*** con un argumento ***arg.***

- Un thread termina su propia ejecución llamando a:

  pthread_exit(value);

- Un thread padre puede esperar a que termine un hijo con:

  pthread_join(tid, value_ptr);

  - ✓ donde ***tid*** es un descriptor y ***value_ptr*** es la dirección de una posición para el valor de retorno (que se llena cuando el hijo llama a exit).

Semáforos con Pthreads

- Los threads pueden sincronizar por semáforos (librería *semaphore.h*).
- Declaración y operaciones con semáforos en Pthreads:
  - ✓ **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.
  - ✓ Existen funciones extras para: wait condicional, obtener el valor de un semáforo y destruir un semáforo (ESTE TIPO DE FUNCIONES EXTRAS NO SE PUEDEN USAR EN LA PRÁCTICA DE LA MATERIA).

Semáforos con Pthreads

Productor / consumidor

- Las funciones de ***Productor*** y ***Consumidor*** serán ejecutadas por threads independientes.
- Acceden a un buffer compartido (***datos***).
- El productor deposita una secuencia de enteros de 1 a ***numItems*** en el buffer.
- El consumidor busca estos valores y los suma.
- Los semáforos ***vacio*** y ***lleno*** garantizan el acceso alternativo de productor y consumidor sobre el buffer.

```
#include <pthread.h>
#include <semaphore.h>
#define SHARED 1

void *Productor(void *);
void *Consumidor(void *);

sem_t  vacio, lleno;
int dato, numItems;
```

```
int main(int argc, char * argv[ ])
{    ……
     sem_init (&vacio, SHARED, 1);
     sem_init (&lleno, SHARED, 0);
     ……
     pthread_create (&pid, &attr,  Productor, NULL);
     pthread_create (&cid, &attr, Consumidor, NULL);
     pthread_join (pid, NULL);
     pthread_join (cid, NULL);
}
```

Semáforos con Pthreads

Productor / consumidor

```
void *Productor (void *arg)
{ int item;
  for (item = 1; item <= numItems; item++)
    {   sem_wait(&vacio);
        dato = item;
        sem_post(&lleno);
     }
   pthreads_exit();
}

void *Consumidor (void *arg)
{   int total = 0, item, aux;
    for (item = 1; item <= numItems; item++)
      { sem_wait(&lleno);
        aux = dato;
        sem_post(&vacio);
        total = total + aux;
      }
    printf("TOTAL:  %d\n", total);
    pthreads_exit();
}
```

Monitores con Pthreads

Variables mutex

- Las secciones críticas se implementan utilizando *mutex_locks* (bloqueo por exclusión mutua).
- Dos estados: *locked* (bloqueado) and *unlocked* (desbloqueado). En cualquier instante, sólo UN thread puede bloquear un *mutex_lock*.
- Para entrar en la SC un Thread debe bloquear el *mutex_lock*. Y cuando sale de la SC debe desbloquear el *mutex_lock*. Todos los *mutex_lock* deben inicializarse como desbloqueados.
- La API Pthreads provee las siguientes funciones para manejar los mutex-locks:

```
int pthread_mutex_lock ( pthread_mutex_t *mutex_lock);
int pthread_mutex_unlock (pthread_mutex_t *mutex_lock);
int pthread_mutex_init (pthread_mutex_t *mutex_lock, const pthread_mutexattr_t *lock_attr);
```

Monitores con Pthreads

Variables Condición

- Podemos utilizar variables de condición para que un thread se autobloquee hasta que se alcance un estado determinado del programa.
- Una variable de condición siempre tiene un mutex asociada a ella.
- Con estas dos herramientas se simulan los monitores: con mutex se hace la exclusión mutua de los mismos, y con las variables condición la sincronización.
- Algunas de las funciones de la API para manejar las variables condición son:

```
int pthread_cond_wait ( pthread_cond_t *cond,  pthread_mutex_t *mutex)
int pthread_cond_signal (pthread_cond_t *cond)
int pthread_cond_broadcast (pthread_cond_t *cond)
```

Monitores con Pthreads

Productor / consumidor

Main de la solución al problema de productores-consumidores.

```
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t task_queue_cond_lock;
int task_available;
...
main()
   { ...
     task_available = 0;
     pthread_init();
     pthread_cond_init(&cond_queue_empty, NULL);
     pthread_cond_init(&cond_queue_full, NULL);
     pthread_mutex_init(&task_queue_cond_lock, NULL);
     ...
   }
```

Monitores con Pthreads

Productor / consumidor

Código para los *productores*.

```
void *producer(void *producer_thread_data)
   { int inserted;

     while (!done())
       { create_task ();
         pthread_mutex_lock (&task_queue_cond_lock);
         while (task_available = = 1)
                pthread_cond_wait (&cond_queue_empty, &task_queue_cond_lock);
          insert_into_queue ();
          task_available = 1;
          pthread_cond_signal (&cond_queue_full);
          pthread_mutex_unlock (&task_queue_cond_lock);
       }
  }
```

Monitores con Pthreads

Productor / consumidor

Código para los *consumidores*.

```
void *consumer(void *consumer_thread_data)
   { while (!done())
        { pthread_mutex_lock (&task_queue_cond_lock);
          while (task_available = = 0)
                pthread_cond_wait (&cond_queue_full, &task_queue_cond_lock);
          my_task = extract_from_queue ();
          task_available = 0;
          pthread_cond_signal (&cond_queue_empty);
          pthread_mutex_unlock (&task_queue_cond_lock);
          process_task (my_task);
        }
   }
```

MPI – Librería para pasaje de mensajes

Extensión de lenguajes secuenciales con bibliotecas específicas

- Una técnica muy utilizada es el desarrollo de bibliotecas de funciones que permiten comunicar/sincronizar procesos, no dependientes de un lenguaje de programación determinado.
- Las soluciones basadas en bibliotecas pueden ser menos eficientes que los lenguajes “reales” de programación concurrente, aunque permiten “agregarse” al código secuencial con bajo costo de desarrollo.
- Las arquitecturas distribuidas han potenciado las soluciones basadas en PVM o MPI, que son básicamente bibliotecas de comunicaciones.
- Los programas MPI usan un estilo SPMD. Cada proceso ejecuta una copia del mismo programa, y puede tomar distintas acciones de acuerdo a su “identidad”. Las instancias interactúan llamando a funciones MPI, que soportan comunicación punto a punto y colectivas.

Conceptos generales

- MPI define una librería estándar para pasaje de mensajes que puede ser empleada desde C o Fortran (y potencialmente desde otros lenguajes).
- El estándar MPI define la sintaxis y la semántica de más de 125 rutinas.
- Hay implementaciones de MPI de la mayoría de los proveedores de hardware.
- Modelo SPMD.
- Todas las rutinas, tipos de datos y constantes en MPI tienen el prefijo “MPI_”. El código de retorno para operaciones terminadas exitosamente es MPI_SUCCESS.
- Básicamente con 6 rutinas podemos escribir programas paralelos basados en pasaje de mensajes: MPI_Init, MPI_Finalize, MPI_Comm_size, MPI_Comm_rank, MPI_Send y MPI_Recv.

Ejemplo

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

```
# include  <mpi.h>
main (INT argc, CHAR *argv [ ] )  {
      INT myid, otherid, size;
      INT length=1, tag=1;
      INT myvalue, othervalue;
      MPI_status  status;

      MPI_Init (&argc, &argv);
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      MPI_Comm_Rank (MPI_COMM_WORLD, &myid);
      IF (myid == 0)  { otherid = 1; myvalue=14;}
      ELSE { otherid=0; myvalue=25; }

      MPI_send (&myvalue, length, MPI_INT, otherid, tag,MPI_COMM_WORLD);
      MPI_recv (&othervalue, length, MPI_INT, MPI_any_source, tag, MPI_COMM_WORLD, &status);
      printf ("process %d received a %d\n", myid, othervalue);
      MPI_Finalize ( );
 }
```

Comunicación punto a punto

- Ejemplo:

```
P0                          P1
a = 100;                    receive(&a, 1, 0)
send(&a, 1, 1);             printf("%d\n", a);
a = 0;
```

- La semántica del SEND requiere que en P1 quede el valor 100 (no 0).
- Para asegurar la semántica del SEND → no devolver el control del Send hasta que el dato a trasmitir esté seguro (Send bloqueante).
- 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

- MPI_Send: rutina básica para enviar datos a otro proceso.

  *MPI_Send (void *buf, int cantidad, MPI_Datatype tipoDato, int destino, int tag, MPI_Comm comunicador)*

  - Valor de Tag entre [0..MPI_TAG_UB].

- 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)*

  - Comodines MPI_ANY_SOURCE y MPI_ANY_TAG.
  - Estructura MPI_Status

    typedef struct MPI_Status { int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR; }

  - MPI_Get_count para obtener la cantidad de elementos recibido.

    *MPI_Get_count (MPI_Status *estado, MPI_Datatype tipoDato, int *cantidad)*

Send y Recv no bloqueante (Isend - Irecv)

- 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)*

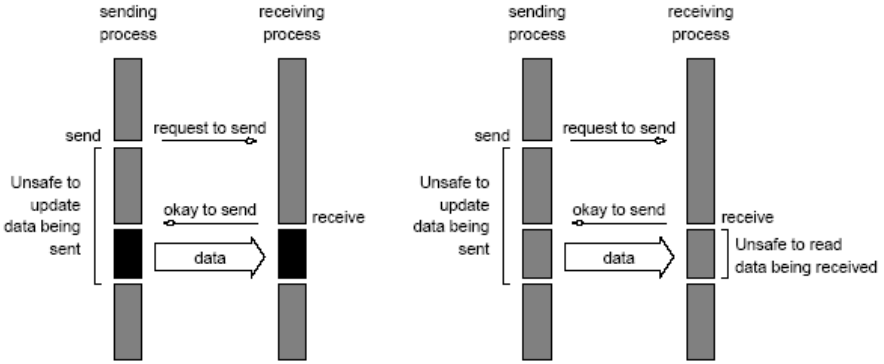
  *MPI_Irecv (void *buf, int cantidad, MPI_Datatype tipoDato, int origen, int tag, MPI_Comm comunicador, MPI_Request *solicitud)*

- 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.

  *MPI_Wait (MPI_Request *solicitud, MPI_Status *estado)*



Indagación por arribo de mensajes

- Información de un mensaje antes de hacer el *Recv* (Origen, Cantidad de elementos, Tag).
- MPI_Probe: bloquea el proceso hasta que llegue un mensaje que cumpla con el origen y el tag.

  *MPI_Probe (int origen, int tag, MPI_Comm comunicador, MPI_Status *estado)*
- MPI_Iprobe: cheqea por el arribo de un mensaje que cumpla con el origen y tag.

  *MPI_Iprobe (int origen, int tag, MPI_Comm comunicador, int *flag, MPI_Status *estado)*
- Comodines en Origen y Tag.

Comunicaciones colectivas

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

- Ventajas del uso de comunicaciones colectivas.