

SISTEMAS DISTRIBUIDOS Y PARALELOS

Carrera: Ingeniería en computación
Facultad de Informática – Universidad Nacional de La Plata



Dr. Adrián Pousa

Modelo de programación sobre memoria compartida

Agenda

2

I. Modelo de programación sobre memoria compartida: Introducción

II. Posix Threads (Pthreads)

I. Gestión de hilos

II. Sincronización (Mutex | Variables Condición | Barreras)

III. Afinidad

III. OpenMP

I. Estructura de control paralela

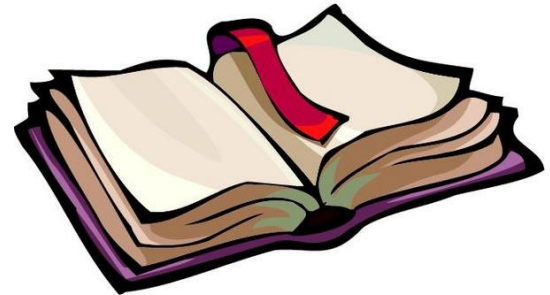
II. Distribución de trabajo entre hilos

III. Gestión de ambiente de datos

IV. Sincronización

V. Funciones y variables de ambiente

VI. Afinidad



Agenda

I. *Modelo de programación sobre memoria compartida: Introducción*

II. *Posix Threads (Pthreads)*

I. *Gestión de hilos*

II. *Sincronización (Mutex | Variables Condición | Barreras)*

III. *Afinidad*

III. *OpenMP*

I. *Estructura de control paralela*

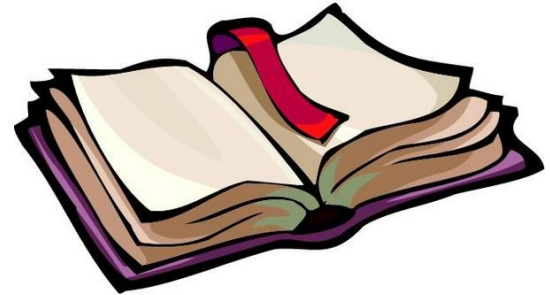
II. *Distribución de trabajo entre hilos*

III. *Gestión de ambiente de datos*

IV. *Sincronización*

V. *Funciones y variables de ambiente*

VI. *Afinidad*



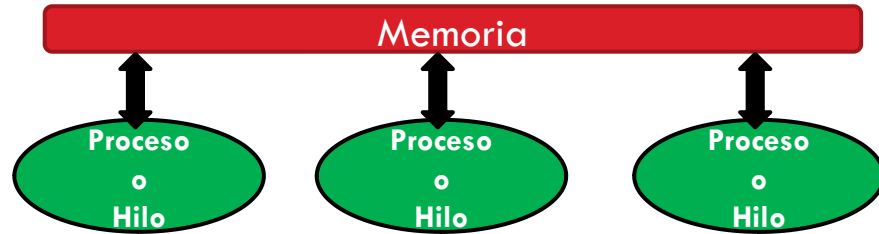
Modelo de programación

4

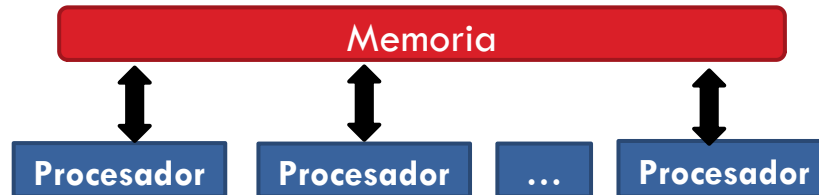
- Paralelismo Implícito.
- Paralelismo Explícito:
 - ▣ Modelo de programación sobre memoria compartida
 - ▣ Modelo de programación sobre memoria distribuida
 - ▣ Modelos híbridos

Modelo de programación sobre memoria compartida

- En un modelo de memoria compartida los procesos o hilos comparten la memoria y se comunican mediante variables.



- Problemas de “interferencia”, necesidad de sincronizar.
- Es posible sólo en **arquitecturas de memoria compartida** con un único espacio de memoria direccionable.



Modelo de programación sobre memoria compartida

6

- Herramientas de desarrollo:
 - ▣ Posix threads (Pthreads)
 - ▣ OpenMP
 - ▣ **Otros:** Cilk, Cuda/OpenCL, Intel Threading Building Blocks (Intel TBB) etc.

Agenda

7

I. *Modelo de programación sobre memoria compartida: Introducción*

II. *Posix Threads (Pthreads)*

I. *Gestión de hilos*

II. *Sincronización (Mutex | Variables Condición | Barreras)*

III. *Afinidad*

III. *OpenMP*

I. *Estructura de control paralela*

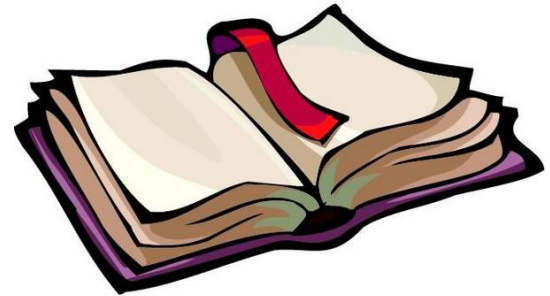
II. *Distribución de trabajo entre hilos*

III. *Gestión de ambiente de datos*

IV. *Sincronización*

V. *Funciones y variables de ambiente*

VI. *Afinidad*



Pthreads

8

- ❑ Pthread (**Posix threads** - estándar Posix 1.0c 1995) API para programación multihilo sobre memoria compartida multiplataforma
- ❑ Posee un conjunto de funciones que permiten:
 - ❑ Gestión de hilos (threads)
 - ❑ Sincronización de los hilos: Mutexes, Variables condición, Barreras
 - ❑ Puede interactuar con la API Semaphore (no es parte del estándar)
- ❑ Incluida en el compilador GCC:
 - ❑ Cabecera de programa C: `#include<pthread.h>`
 - ❑ Compilación: `gcc -pthread fuente.c -o ejecutable`
- ❑ <https://computing.llnl.gov/tutorials/pthreads/>

Agenda

I. *Modelo de programación sobre memoria compartida: Introducción*

II. *Posix Threads (Pthreads)*

I. *Gestión de hilos*

II. *Sincronización (Mutex | Variables Condición | Barreras)*

III. *Afinidad*

III. *OpenMP*

I. *Estructura de control paralela*

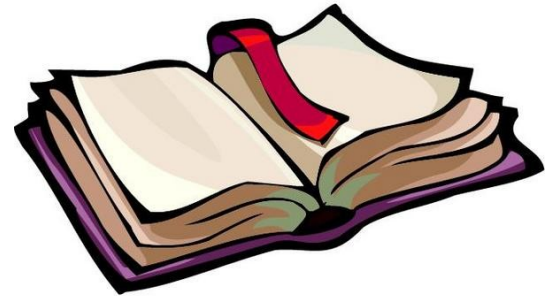
II. *Distribución de trabajo entre hilos*

III. *Gestión de ambiente de datos*

IV. *Sincronización*

V. *Funciones y variables de ambiente*

VI. *Afinidad*



Pthreads – Modelo de programa pthreads

10

□ 4 pasos para utilizar hilos:

- 1) Definirlos
- 2) Crearlos
- 3) Implementar la función de comportamiento del hilo.
- 4) Esperar que el hilo termine

```
#include<pthread.h>
...
//3) Función de comportamiento del hilo
void* f(void* arg){
    ...
    pthread_exit(NULL);
}
...
int main(int argc, char*argv[]){
    pthread_t miHilo; //1) Definición
    ...
    pthread_create(&miHilo,&attr,&f,&arg); //2) Creación
    ...
    pthread_join(&miHilo,NULL); //4) Espera finalización
    return 0;
}
```

Pthreads – Definición de hilos

11

- Los hilos pueden definirse individualmente:

```
pthread_t miHilo;
```

- O como arreglos de hilos:

```
pthread_t misHilos[N];
```

Pthreads – Creación de hilos

12

- Los hilos definidos se crean mediante la función:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);
```

Parámetros:

- **thread:** hilo definido con pthread_t.
- **attr:** atributos de configuración del hilo a crear.
- **start_routine:** puntero a la función que implementa el comportamiento del hilo.
- **arg:** argumento pasado como parámetro a la función anterior.

Valor de retorno:

pthread_create retorna cero si tiene éxito, sino retorna un código de error.

Si **pthread_create** tiene éxito, el hilo creado se ejecuta inmediatamente.

Pthreads – Creación de hilos - Atributos

13

- El parámetro **attr** puede ser **NULL** y el hilo se creará con los atributos por defecto
- Pasos para cambiar atributos:

- 1) Definirlos
- 2) Inicializarlos
- 3) Modificarlos
- 4) Invocar create
- 5) Destruirlos

```
...
int main(int argc, char*argv[]){
    pthread_t miHilo;
    pthread_attr_t *attr; //1) Definición
    ...
    pthread_attr_init(&attr); //2) Inicialización
    //3) Funciones para modificación de atributos
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    pthread_attr_schedpolicy(&attr, SCHED_OTHER);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    ...
    pthread_create(&miHilo, &attr, &f, &arg); //4) Create con attr
    ...
    pthread_attr_destroy(&attr); //5) Destrucción
    return 0;
}
```

Pthreads – Creación de hilos - Comportamiento

14

- El prototipo para la función que implementa el comportamiento del hilo es:

```
void* funcion(void *arg) ;
```

- En la función de creación:
 - Hilos con el mismo comportamiento: se pasa el mismo puntero a función
 - Hilos con distinto comportamiento: puntero a una función distinta para cada hilo

```
void* funcionA(void *arg){  
...  
}
```

```
void* funcionB(void *arg){  
...  
}
```

```
pthread_t T1,T2,T3;  
  
pthread_create(&T1, &attr, &funcionA,...) ;  
pthread_create(&T2, &attr, &funcionA,...) ;  
pthread_create(&T3, &attr, &funcionB,...) ;
```

Pthreads – Creación de hilos - Argumentos

15

- La función que implementa el comportamiento de los hilos tiene un ÚNICO argumento (*void).
- El argumento puede ser:
 - ▣ Un valor de un tipo estándar
 - ▣ Una estructura (registro): NO HABITUAL
- En ambos casos es necesario realizar los castings correspondientes.

Pthreads – Creación de hilos - Argumentos

16

- Ejemplo: pasando una variable de tipo estándar.

```
void* funcion(void *arg){  
    int x_local=*(int*)arg;  
    ...  
}  
...  
int main(int argc, char* argv[]){  
    int x=1;  
    ...  
    pthread_create(&miHilo,&attr,&funcion,(void*)&x);  
    ...  
}
```


Pthreads – Creación de hilos - Argumentos

17

- Ejemplo: pasando una estructura.

```
typedef struct str_punto{
    int x;
    int y;
} punto_t;

void* funcion(void *arg){
    punto_t punto_local=(punto_t*) arg;
    ...
}

...
int main(int argc, char* argv[]){
    punto_t punto;
    ...
    pthread_create(&miHilo,&attr,&funcion,(void*)&punto);
    ...
}
```

Pthreads – Creación de hilos - Argumentos

18

- ❑ Pthreads no tiene una forma de identificar los hilos.
- ❑ Generalmente, el programador debe pasar un identificador como parámetro a la función que implementa el comportamiento del hilo.

```
...
void* funcion(void *arg){
    int id=*(int*)arg;
    ...
}

...
int main(int argc, char* argv[]){
    pthread_t misHilos[T];
    ...
    for(int id=0;id<T;id++){
        pthread_create(&misHilos[id], &attr, &funcion, (void*)&id);
    }
    ...
}
```

Pthreads – Creación de hilos - Argumentos

19

Pero este código puede tener problemas!!!

```
for(int id=0;id<T;id++)  
    pthread_create(&miHilo,&attr,&funcion,(void*)&id);
```

- El argumento *id* es un puntero al índice del *for*, en el *for* podría modificarse el valor de la variable *id* antes que el hilo creado la lea, como consecuencia puede haber más de un hilo con el mismo *id*. *Una solución es:*

```
int main(int argc, char* argv){  
    pthread_t misHilos[T];  
    int threads_ids[T];  
    ...  
    for(int id=0;id<T;id++){  
        threads_ids[id]=id;  
        pthread_create(&misHilos[id],&attr,&funcion,(void*)&threads_ids[id]);  
    }  
    ...  
}
```

Pthreads – Creación de hilos - Argumentos

20

- Alternativa **IBM** sin utilizar un array:

```
int main(int argc, char* argv[]){  
    pthread_t misHilos[T];  
    ...  
    for(int id=0;id<T;id++){  
        pthread_create(&misHilos[id],&attr,&funcion, (void*)id);  
    }  
    ...  
}
```

id es un valor no un puntero!!!
Es pasarle un entero y decirle que es una dirección de memoria.

Pthreads – Finalización de hilos

- Al finalizar el código de cada hilo se invoca la función:

```
void pthread_exit(void *value_ptr);
```

- Esta función termina la ejecución del hilo.

Parámetros:

- **value_ptr:** es un valor de retorno que puede ser NULL o un valor que será enviado a cualquier proceso/hilo que espere por su finalización (JOIN).

```
...  
void* funcion(void *arg){  
...  
    pthread_exit(NULL);  
}  
...
```

```
...  
void* funcion(void *arg){  
    int ret = 2;  
...  
    pthread_exit(&ret);  
}  
...
```

Pthreads – Join

22

- La función `pthread_exit` suele usarse en conjunto con la función JOIN:

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Parámetros:

- **thread**: hilo al que debe esperar el hilo que ejecute `pthread_join`.
- **value_ptr**: valor retornado por `pthread_exit` del **thread** recibido como parámetro. Puede ir NULL si no es necesario recibir ningún dato desde `pthread_exit` de **thread**.

Valor de retorno:

pthread_join retorna cero si tiene éxito, sino retorna un código de error.

Pthreads – Join

23

- El programa C que se ejecuta y crea los hilos es un proceso que actúa como un hilo más (main).
- El main, luego de crear los hilos, hace un join esperando que estos finalicen.

```
void* funcionT1(void *arg){  
...  
    pthread_exit(NULL);  
}
```

```
void* funcionT2(void *arg){  
    int ret = 2;  
...  
    pthread_exit(&ret);  
}
```

```
int main(int argc, char* argv[]){  
    pthread_t T1,T2;  
    int valorT2;  
    pthread_create(&T1,...,&funcionT1,...);  
    pthread_create(&T2,...,&funcionT2,...);  
    pthread_join(T1,NULL);  
    pthread_join(T2, (void**)&valorT2);  
...  
}
```

Pthreads – Join

24

- Ejemplo utilizando un arreglo de hilos:

```
void* funcionT2(void *arg){
    int ret = 2;
    ...
    pthread_exit(&ret);
}

...
int main(int argc, char* argv[]){
    pthread_t misThreads[N];
    for(i=0;i<N;i++){
        ...
        pthread_create(&misThreads[i],...,&funcion,...);
    }
    for(i=0;i<N;i++){
        pthread_join(misThreads[i], (void**) &valorRet);
    }
    ...
}
```


Agenda

I. *Modelo de programación sobre memoria compartida: Introducción*

II. *Posix Threads (Pthreads)*

I. *Gestión de hilos*

II. *Sincronización (Mutex | Variables Condición | Barreras)*

III. *Afinidad*

III. *OpenMP*

I. *Estructura de control paralela*

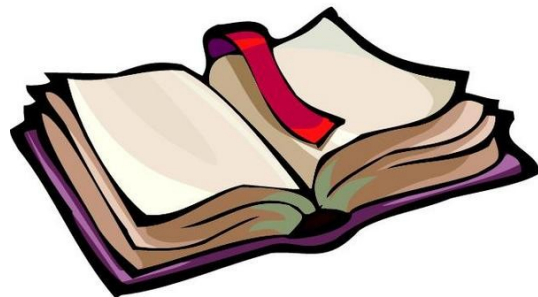
II. *Distribución de trabajo entre hilos*

III. *Gestión de ambiente de datos*

IV. *Sincronización*

V. *Funciones y variables de ambiente*

VI. *Afinidad*



Pthreads – Sincronización

26

- Pthreads permite la sincronización mediante mecanismos como:
 - ▣ Mutexes
 - ▣ Variables condición
 - ▣ Barreras

- Es posible implementar semáforos mediante la biblioteca semaphore pero no están definidos en el estándar.

Agenda

I. *Modelo de programación sobre memoria compartida: Introducción*

II. *Posix Threads (Pthreads)*

I. *Gestión de hilos*

II. *Sincronización (Mutex | Variables Condición | Barreras)*

III. *Afinidad*

III. *OpenMP*

I. *Estructura de control paralela*

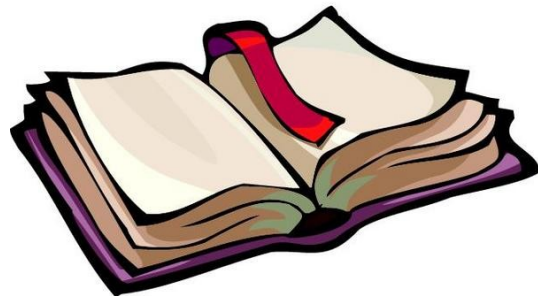
II. *Distribución de trabajo entre hilos*

III. *Gestión de ambiente de datos*

IV. *Sincronización*

V. *Funciones y variables de ambiente*

VI. *Afinidad*



Pthreads – Sincronización - Mutexes

28

- Los mutexes, abreviación de mutual exclusion, se utilizan para sincronización por exclusión mutua.

- Los mutexes tienen dos posibles estados:
 - ▣ Bloqueado/Locked: apropiado por un hilo.
 - ▣ Desbloqueado/Unlocked: libre.

- Si un mutex está libre sólo puede ser apropiado por un único hilo.

Pthreads – Sincronización - Mutexes

29

□ Los pasos para utilizar mutex son:

- 1) Definirlos (globales)
- 2) Inicializarlos
- 3) Utilizarlos
- 4) Destruirlos

```
#include<pthread.h>

pthread_mutex_t miMutex; //1) Definición global
...
void* f(void* arg){
    //3) Uso
    pthread_mutex_lock(&miMutex);
    //Region critica
    pthread_mutex_unlock(&miMutex);
}
...
int main(int argc, char*argv[]){
    ...
    pthread_mutex_init(miMutex, &mutex_attr); //2) Inicialización
    pthread_create(&miHilo,&attr,&f,&arg);
    ...
    pthread_mutex_destroy(&miMutex); //4) Destrucción
}
```

Pthreads – Sincronización - Definición de mutexes

- Los mutexes pueden definirse individualmente:

```
pthread_mutex_t miMutex;
```

- O como arreglos de mutexes:

```
pthread_mutexes_t misMutexes[N];
```

- Visibilidad:

- ▣ Para que los mutexes puedan ser usados deben ser accesibles por todos los hilos
- ▣ Lo correcto es definirlos como variables compartidas (fuera del main o en archivos separados)

Pthreads – Sincronización – Inicialización de mutexes

- ❑ Los mutexes deben inicializarse antes de ser usados.
- ❑ Dos formas de hacerlo:

- ❑ Estática:

```
pthread_mutex_t miMutex = PTHREAD_MUTEX_INITIALIZER;
```

- ❑ Dinámica, mediante la función:

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *m_attr);
```

El parámetro **m_attr** permite personalizar los atributos del mutex.

Puede ser **NULL** o una estructura de tipo `pthread_mutexattr_t` que contiene tres atributos:

Protocol: protocolo usado para prevenir inversión de prioridades.

Prioc ceiling: especifica el límite de prioridad de un mutex.

Process-shared: especifica el uso compartido de un mutex.

Pthreads – Sincronización – Uso de mutexes

- Existen dos funciones básicas para utilizar mutexes:

- ▣ Adquirir el mutex:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Si el mutex está desbloqueado el propietario será el hilo que invoca a la función. Cualquier otro hilo que invoque a la función se quedará dormido.

- ▣ Liberar el mutex:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Ambas funciones retornan cero si tienen éxito, sino retornan un código de error.

Pthreads – Sincronización – Uso de mutexes

33

- Adicionalmente, la función:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- Si el hilo **puede** adquirir el mutex será el propietario y la función retorna 0
- Si el hilo **no puede** adquirir el mutex continúa su ejecución y la función no retorna 0

- El uso correcto de esta función es:

```
if (pthread_mutex_trylock(&miMutex) == 0) {  
    ...  
    pthread_mutex_unlock(&miMutex);  
}
```

Pthreads – Sincronización – Destrucción de mutexes

- Una vez que un mutex deja de usarse debe ser destruido con la función:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

pthread_mutex_destroy retorna cero si tienen éxito, sino retorna un código de error.

Agenda

I. *Modelo de programación sobre memoria compartida: Introducción*

II. *Posix Threads (Pthreads)*

I. *Gestión de hilos*

II. *Sincronización (Mutex | Variables Condición | Barreras)*

III. *Afinidad*

III. *OpenMP*

I. *Estructura de control paralela*

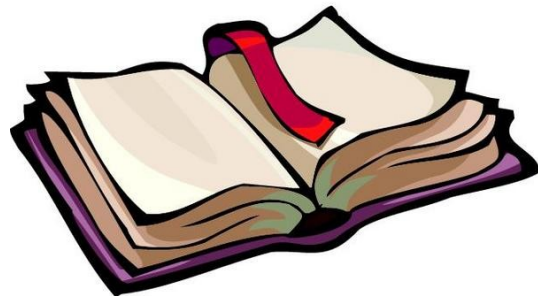
II. *Distribución de trabajo entre hilos*

III. *Gestión de ambiente de datos*

IV. *Sincronización*

V. *Funciones y variables de ambiente*

VI. *Afinidad*



Pthreads – Sincronización – Variables condición

36

- ❑ Las variables condición se utilizan para sincronización por condición.
- ❑ Permiten detener la ejecución de un hilo a la espera de la ocurrencia de alguna condición.
- ❑ Cuando esa condición se cumple, algún hilo enviará una señal al hilo dormido para que continúe su ejecución.
- ❑ Cada variable condición tiene asociada una cola de espera.
- ❑ Se deben utilizar en conjunto con un *Mutex*.

Pthreads – Sincronización – Variables condición

37

□ Los pasos para utilizarlas son:

- 1) Definirlas (globales)
- 2) Inicializarlas
- 3) Utilizarlas
- 4) Destruirlas

```
#include<pthread.h>
pthread_cond_t c; //1) Definición
```

```
void* f1(void* arg){
    //3) Uso
    pthread_mutex_lock(&mutex);
    ...
    pthread_cond_wait(&c,&mutex);
    ...
    pthread_mutex_unlock(&mutex);
}
```

```
void* f2(void* arg){
    //3) Uso
    pthread_mutex_lock(&mutex);
    ...
    pthread_cond_signal(&c);
    ...
    pthread_mutex_unlock(&mutex);
}
```

```
int main(int argc, char*argv[]){
    ...
    pthread_cond_init(&c,&cond_attr); //2) Inicialización
    pthread_create(&miHilo1,&attr1,&f1,&arg1);
    pthread_create(&miHilo2,&attr2,&f2,&arg2);
    ...
    pthread_cond_destroy(&c); //4) Destrucción
}
```

Pthreads – Sincronización

Variables condición - Definición

38

- Las variables condición pueden definirse individualmente:

```
pthread_cond_t c;
```

- O como arreglos de mutexes:

```
pthread_cond_t cs[N];
```

- Visibilidad:

- ▣ Para que puedan ser usadas deben ser accesibles por todos los hilos
- ▣ Lo correcto es definir las como variables compartidas (fuera del main o en archivos separados)

Pthreads – Sincronización

Variables condición - Inicialización

39

- ❑ Las variables condición deben inicializarse antes de ser usadas.
- ❑ Dos formas de hacerlo:

- ❑ Estática:

```
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
```

- ❑ Dinámica, mediante la función:

```
int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr);
```

El parámetro **attr** permite personalizar los atributos de la variable condición.

Puede ser NULL o una estructura de tipo `pthread_condattr_t` que permite limitar el alcance de la variable condición y solo puede modificarse mediante funciones:

`pthread_condattr_getpshared`

`pthread_condattr_setpshared`

etc...

Pthreads – Sincronización

Variables condición - Uso

40

- Existen tres funciones básicas aplicables a variables condición:

- Dormir un hilo:

```
int pthread_cond_wait(&c, &mutex)
```

- Despertar un hilo:

```
int pthread_cond_signal(&c)
```

- Despertar todos los hilos:

```
int pthread_cond_broadcast(&c)
```

Todas estas funciones retornan cero si tienen éxito, sino retornan un código de error.

Pthreads – Sincronización

Variables condición - Uso

41

- La función `pthread_cond_wait` sigue el siguiente prototipo:

```
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t * mutex);
```

- La función necesita un mutex asociado a la variable condición.
- El uso habitual de `pthread_cond_wait` es:

```
pthread_mutex_lock(&mutex);  
...  
pthread_cond_wait(&c, &mutex);  
...  
pthread_mutex_unlock(&mutex);
```

Las función `pthread_cond_wait` libera el mutex automáticamente y pone el hilo a dormir. Cuando el hilo se despierte tomará el mutex automáticamente.

Pthreads – Sincronización

Variables condición - Uso

42

- Las función `pthread_cond_signal` (que despierta un solo hilo) y `pthread_cond_broadcast` (que despierta todos los hilos) siguen los prototipos:

```
int pthread_cond_signal(pthread_cond_t *cond);
```

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- Aunque no tienen asociado un mutex, es habitual usarlas en conjunto con el mutex asociado a la función `pthread_cond_wait`:

```
pthread_mutex_lock(&mutex);  
...  
pthread_cond_signal(&c);  
...  
pthread_mutex_unlock(&mutex);
```

```
pthread_mutex_lock(&mutex);  
...  
pthread_cond_broadcast(&c);  
...  
pthread_mutex_unlock(&mutex);
```

Pthreads – Sincronización

Variables condición - Destrucción

43

- Una vez que una variable condición deja de usarse debe destruirse con la función:

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

pthread_cond_destroy retorna cero si tienen éxito, sino retorna un código de error.

Agenda

44

I. *Modelo de programación sobre memoria compartida: Introducción*

II. *Posix Threads (Pthreads)*

I. *Gestión de hilos*

II. *Sincronización (Mutex | Variables Condición | Barreras)*

III. *Afinidad*

III. *OpenMP*

I. *Estructura de control paralela*

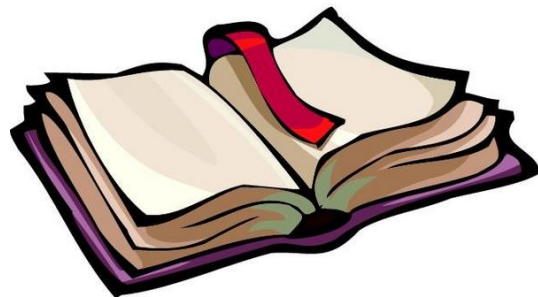
II. *Distribución de trabajo entre hilos*

III. *Gestión de ambiente de datos*

IV. *Sincronización*

V. *Funciones y variables de ambiente*

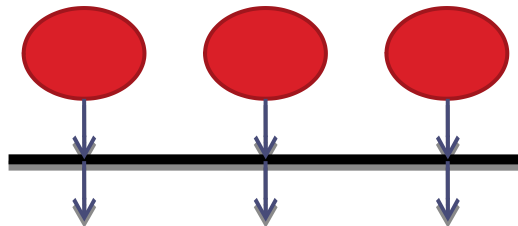
VI. *Afinidad*



Pthreads – Sincronización – Barreras

45

- Una herramienta de sincronización muy común son las barreras.
- Una barrera hace que un conjunto de procesos se esperen para poder continuar su ejecución.



- Pthreads provee de una implementación sencilla y eficiente de las barreras y evita que el programador tenga que implementarlas.

Pthreads – Sincronización - Barreras

46

□ Los pasos para utilizarlas son:

- 1) Definirlas (globales)
- 2) Inicializarlas
- 3) Utilizarlas
- 4) Destruirlas

```
#include<pthread.h>

pthread_barrier_t barrera; //1) Definición
...
void* f(void* arg){
    //3) Uso
    pthread_barrier_wait(&barrera);
...
}
...
int main(int argc, char*argv[]){
    ...
    pthread_barrier_init(&barrera, &b_attr, 3); //2) Inicialización
    pthread_create(&miHilo1,&attr,&f,&arg);
    pthread_create(&miHilo2,&attr,&f,&arg);
    pthread_create(&miHilo3,&attr,&f,&arg);
    ...
    pthread_barrier_destroy(&barrera); //4) Destrucción
}
```

Pthreads – Sincronización

Barreras – Definición e inicialización

47

- Las barreras se definen como:

```
pthread_barrier_t barrera;
```

- Deben inicializarse antes de ser usadas:

```
int pthread_barrier_init(pthread_barrier_t * b, const pthread_barrierattr_t *attr, unsigned count);
```

El parámetro **attr** permite personalizar los atributos de la barrera.

Puede ser NULL o una estructura de tipo `pthread_barrierattr_t` que permite definir atributos con las características de la barrera y solo puede modificarse mediante funciones:

`pthread_barrierattr_getpshared`

`pthread_barrierattr_setpshared`

El parámetro **count** indica el número de hilos que deben llegar a la barrera para continuar.

Pthreads – Sincronización

Barreras – Uso y destrucción

48

- Cuando un hilo llega a la barrera debe ejecutar la función:

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

- Si el número de hilos en la barrera es menor al especificado en el parámetro count de la función de inicialización, el hilo se dormirá.
 - En caso contrario todos los hilos dormidos en la barrera continuarán la ejecución.
- Cuando la barrera ya no se utilice se debe destruir:

```
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

Ambas funciones retornan cero si tienen éxito, sino retornan un código de error.

Agenda

49

I. *Modelo de programación sobre memoria compartida: Introducción*

II. *Posix Threads (Pthreads)*

I. *Gestión de hilos*

II. *Sincronización (Mutex | Variables Condición | Barreras)*

III. *Afinidad*

III. *OpenMP*

I. *Estructura de control paralela*

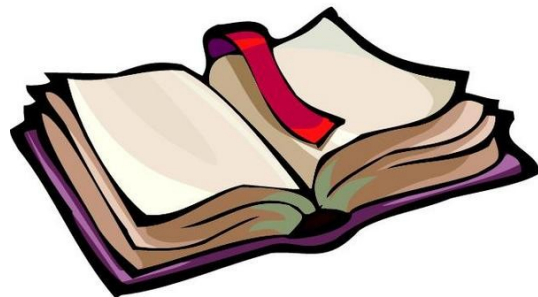
II. *Distribución de trabajo entre hilos*

III. *Gestión de ambiente de datos*

IV. *Sincronización*

V. *Funciones y variables de ambiente*

VI. *Afinidad*



Pthreads – Afinidad

50

Afinidad: elegir el core donde debe ejecutar el hilo

¿Por qué usar afinidad?

- ▣ Suponer una máquina con varios procesadores, cada uno con varios cores
- ▣ Aplicaciones intensivas en memoria pueden obtener mejor rendimiento si se ejecutan con menos hilos que los cores disponibles, esto debido a los fallos de caché
- ▣ Se debe asegurar que dos o más hilos no ejecuten sobre el mismo procesador

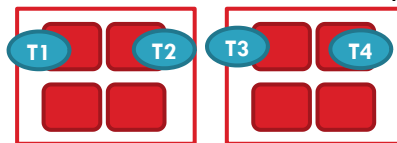
2 Procesadores (4 cores c/u)



**Varios hilos al mismo
procesador**



2 Procesadores (4 cores c/u)



**Distribución equilibrada entre
procesadores**

Pthreads – Afinidad

51

- Para utilizar afinidad es necesario incluir al inicio del archivo las siguientes líneas:

```
#define _GNU_SOURCE
#include<sched.h>
```

- Luego, hacemos afinidad utilizando la función:

```
int pthread_setaffinity_np(pthread_t thread, size_t cpusetsize, const cpu_set_t *cpuset)
```

Parámetros:

- **thread:** hilo al que debe cambiarse la afinidad.
- **cpusetsize:** tamaño en bytes del CPU Set.
- **cpuset:** conjunto de cores de la arquitectura.

Valor de retorno:

pthread_setaffinity_np retorna cero si tiene éxito, sino retorna un código de error.

Pthreads – Afinidad

52

- El parámetro **cpuset** es de tipo `cpu_set_t`, una estructura similar a:

cpuset	0	1	0	1	0	1	0	1
Core ID	0	1	2	3	4	5	6	7

En este ejemplo el hilo puede ejecutar en los cores impares

- Las variables de tipo `cpu_set_t` no pueden modificarse directamente, para esto se utiliza la macro:

```
CPU_SET(int cpu, cpu_set_t *set)
```

Pthreads – Afinidad

53

- La función `pthread_setaffinity_np` puede invocarse en dos lugares distintos del código:
 - ▣ Por el proceso que crea el hilo
 - ▣ Por el propio hilo

Pthreads – Afinidad

54

- Por el proceso que crea el hilo:

```
#define _GNU_SOURCE
#include<pthread.h>
#include<sched.h>

...
int main(int argc, char* argv[]){
    pthread_t hilo; int idHilo=1;
    cpu_set_t mask;

    pthread_create(&hilo, NULL, &funcionHilo, (void*)&idHilo);
    CPU_ZERO(&mask); //Pone en cero la mascara
    CPU_SET(3, &mask); // Pone en uno el bit que representa al procesador 3 en la máscara
    pthread_setaffinity_np(hilo, sizeof(cpu_set_t), &mask); //Cambia la afinidad al hilo
    pthread_join(hilo, NULL);

    ...
}
```

Pthreads – Afinidad

55

- Por el propio hilo:

```
#define _GNU_SOURCE
#include<pthread.h>
#include<sched.h>

void* funcionHilo(void *arg){
    pthread_t hilo=pthread_self(); //Obtiene el descriptor del hilo
    CPU_ZERO(&mask); //Pone en cero la mascara
    CPU_SET(3, &mask); // Pone en uno el bit que representa al procesador 3 en la máscara
    pthread_setaffinity_np(hilo, sizeof(cpu_set_t), &mask); //Le asigna la máscara al hilo
    ...
}

int main(int argc, char* argv[]){
    pthread_t hilo; int idHilo=1;

    pthread_create(&hilo, NULL, &funcionHilo, (void*)&idHilo);
    ...
}
```

Pthreads – Afinidad

- Cuando se utiliza afinidad para distribuir hilos entre los distintos cores se debe tener en cuenta la identificación de los cores que hace el Sistema Operativo.
- Intuitivamente podemos suponer la siguiente identificación:



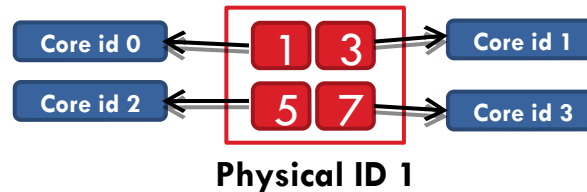
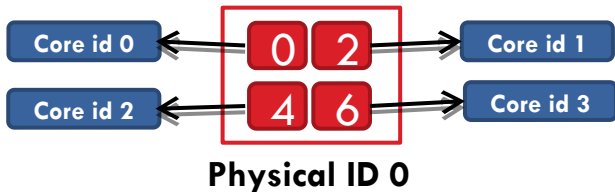
- Sin embargo, cada sistemas operativo y cada distribución de Linux los identifican de manera diferente.
- En Linux, esta identificación podemos obtenerla mediante el comando:

```
cat /proc/cpuinfo
```


Pthreads – Afinidad

57

- El recuadro muestra una salida simplificada de **cpuinfo** de una máquina con dos procesadores quad-core.
- Se pueden observar tres identificadores:
 - **Processor:** identifica la unidad de procesamiento. Numeradas de 0 a 7
 - **Physical id:** identifica el socket del Processor. Numerados 0 y 1
 - **Core id:** identifica el core dentro de ese socket. Numerados 0 a 3 por socket
- La salida nos dice que la topología es la siguiente:



```
processor : 0
physical id : 0
core id : 0
processor : 1
physical id : 1
core id : 0
processor : 2
physical id : 0
core id : 2
processor : 3
physical id : 1
core id : 2
processor : 4
physical id : 0
core id : 1
processor : 5
physical id : 1
core id : 1
processor : 6
physical id : 0
core id : 3
processor : 7
physical id : 1
core id : 3
```

Agenda

58

I. *Modelo de programación sobre memoria compartida: Introducción*

II. *Posix Threads (Pthreads)*

I. *Gestión de hilos*

II. *Sincronización (Mutex | Variables Condición | Barreras)*

III. *Afinidad*

III. *OpenMP*

I. *Estructura de control paralela*

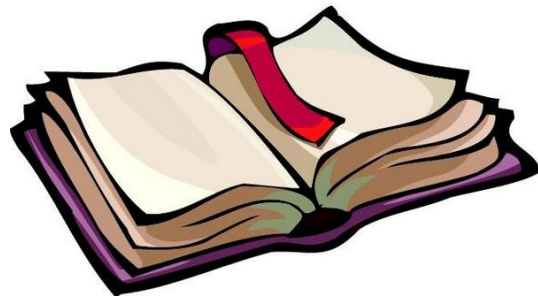
II. *Distribución de trabajo entre hilos*

III. *Gestión de ambiente de datos*

IV. *Sincronización*

V. *Funciones y variables de ambiente*

VI. *Afinidad*



OpenMP

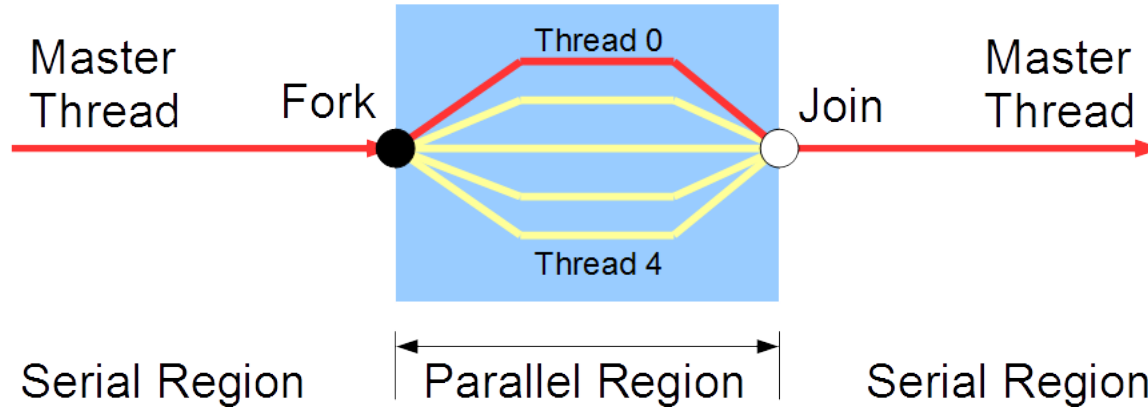
59

- ❑ API para programación paralela sobre memoria compartida multiplataforma
- ❑ Se basa en incluir directivas para los distintos lenguajes
- ❑ Implementada para C, C++ y Fortran
- ❑ Incluida en GCC:
 - ▣ Cabecera de programa: `#include<omp.h>`
 - ▣ Compilación: `gcc -fopenmp fuente.c -o ejecutable`
- ❑ www.openmp.org

OpenMP

60

- OpenMP sigue el modelo **fork-join**: una tarea se divide en T tareas (**fork**), cada una ejecuta su parte en forma paralela y luego se esperan (**join**) en un punto a partir del cual se continua la ejecución secuencial.



□ Se conoce como **Core Elements** a los constructores para:

- Estructura de control paralela o control de flujo (creación de hilos)
- Trabajo compartido - Distribución de trabajo entre hilos
- Gestión de ambiente de datos
- Sincronización
- Rutinas a nivel de usuario y variables de ambiente

OpenMP

62

Core elements

Estructura de control paralela

Flujo de control de programa

Directivas:
parallel

Trabajo compartido

Distribución de trabajo entre hilos

Directivas:
do/parallel do
section
single
master
schedule

Ambiente de datos

Alcance de variables

Directivas:
shared
private

Sincronización

Coordina la ejecución de hilos

Directivas:
critical
atomic
barrier

Funciones Variables

Coordina la ejecución de hilos

Funciones:
omp_get_thread_num()
omp_set_num_threads()
omp_get_num_threads()

Variables:
OMP_NUM_THREADS
OMP_SCHEDULE

Agenda

I. *Modelo de programación sobre memoria compartida: Introducción*

II. *Posix Threads (Pthreads)*

I. *Gestión de hilos*

II. *Sincronización (Mutex | Variables Condición | Barreras)*

III. *Afinidad*

III. *OpenMP*

I. *Estructura de control paralela*

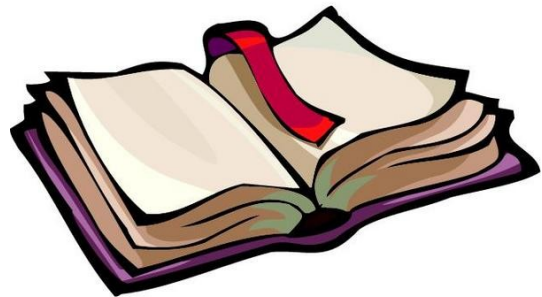
II. *Distribución de trabajo entre hilos*

III. *Gestión de ambiente de datos*

IV. *Sincronización*

V. *Funciones y variables de ambiente*

VI. *Afinidad*



OpenMP - Estructura de control paralela

- En el caso particular del lenguaje C, OpenMP utiliza la directiva **#pragma**
- Las directivas **#pragma** son específicas del compilador. Su forma de uso es:

```
#pragma instrucción
```

- Permiten pasar opciones al compilador. Por ejemplo:
 - ▣ Suprimir un mensaje de error específico
 - ▣ Y en el caso de OpenMP crear hilos
- Si el compilador encuentra **#pragma** con una instrucción desconocida la ignora (sin error)
- La directiva **#pragma** se combina con otras directivas para lograr la funcionalidad deseada. La sintaxis básica para el lenguaje C/C++ es:

```
# pragma omp <directiva> [cláusula [ , ...] ...]
```


OpenMP – Estructura de control paralela

65

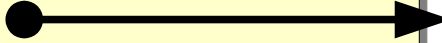
- ❑ OpenMP Hello world!!!
- ❑ El pragma **omp parallel** se usa para crear hilos y establecer el flujo de ejecución
- ❑ Cada hilo ejecutará el código encerrado entre llaves

```
int main(void) {  
    #pragma omp parallel  
    {  
        printf("Hello, world.\n");  
    }  
    return 0;  
}
```

OpenMP – Estructura de control paralela

66

```
int main(void) {  
    "Código secuencial"  
    #pragma omp parallel{  
        "Código paralelo"  
    }  
    "Código secuencial"  
    #pragma omp parallel{  
        "Código paralelo"  
    }  
    "Código secuencial"  
    return 0;  
}
```

A black arrow points from a black dot on the first "Código secuencial" line of the code to the explanatory text box on the right.

El programa inicialmente comienza con el Master thread (ID=0) ejecutando en forma secuencial.

OpenMP – Estructura de control paralela

67

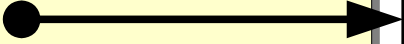
```
int main(void) {  
    "Código secuencial"  
    #pragma omp parallel{  
        "Código paralelo"  
    }  
    "Código secuencial"  
    #pragma omp parallel{  
        "Código paralelo"  
    }  
    "Código secuencial"  
    return 0;  
}
```

Cuando encuentra por primera vez la
directiva **#pragma** crea T hilos.
ID=0..T-1.
(FORK)

OpenMP – Estructura de control paralela

68

```
int main(void) {  
    "Código secuencial"  
    #pragma omp parallel{  
        "Código paralelo"  
    }  
    "Código secuencial"  
    #pragma omp parallel{  
        "Código paralelo"  
    }  
    "Código secuencial"  
    return 0;  
}
```

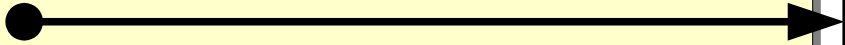


Cada hilo ejecuta en forma paralela el código entre llaves.

OpenMP – Estructura de control paralela

69

```
int main(void) {  
    "Código secuencial"  
    #pragma omp parallel{  
        "Código paralelo"  
    }  
    "Código secuencial"  
    #pragma omp parallel{  
        "Código paralelo"  
    }  
    "Código secuencial"  
    return 0;  
}
```

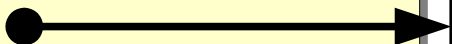


Cada hilo, al finalizar, debe esperar a que el resto termine su ejecución. Se produce una barrera implícita.
(JOIN)

OpenMP – Estructura de control paralela

70

```
int main(void) {  
    "Código secuencial"  
    #pragma omp parallel{  
        "Código paralelo"  
    }  
    "Código secuencial"  
    #pragma omp parallel{  
        "Código paralelo"  
    }  
    "Código secuencial"  
    return 0;  
}
```

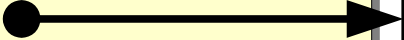


El Master thread continua la ejecución de forma secuencial.
Los hilos restantes se van a dormir.

OpenMP – Estructura de control paralela

71

```
int main(void) {  
    "Código secuencial"  
    #pragma omp parallel{  
        "Código paralelo"  
    }  
    "Código secuencial"  
    #pragma omp parallel{  
        "Código paralelo"  
    }  
    "Código secuencial"  
    return 0;  
}
```



Cuando se encuentra la siguiente directiva
se despierta a los hilos.
(FORK)

**Para el sistema operativo el costo
de crear hilos una y otra vez es
mayor que dormirlos y
despertarlos.**

OpenMP – Estructura de control paralela

72

- Por defecto, OpenMP determina la cantidad de cores de la arquitectura y crea un hilo por cada core.
- Tres formas de indicarle a OpenMP la cantidad de hilos a crear:

- La función **omp_set_num_threads**:

```
omp_set_num_threads(4); //Crea 4 hilos
#pragma omp parallel
...
```

- La variable de entorno **OMP_NUM_THREADS**: **export OMP_NUM_THREADS=4**

La directiva num_threads:

```
#pragma omp parallel num_threads(4) //Crea 4 hilos
...
```


OpenMP – Estructura de control paralela

73

- **Anidamiento:** se crean algunos hilos y estos a su vez vuelven a crear hilos.
- Ejemplo: se crean 2 hilos, cada uno crea 4 hilos. (8 hilos en total)

```
omp_set_nested(1); // 1:anidamiento habilitado, 0: deshabilitado
#pragma omp parallel num_threads(2)
{
    printf("ID hilo externo:%d\n",omp_get_thread_num());
    #pragma omp parallel num_threads(4)
    {
        printf("ID hilo interno:%d\n",omp_get_thread_num());
    }
}
```

**Sin `omp_set_nested(1)` el número total de hilos será 2!!!
La creación de los 4 hilos internos se ignora!!!.**

Posible salida:

ID hilo externo:0
ID hilo externo:1
ID hilo interno:0
ID hilo interno:3
ID hilo interno:1
ID hilo interno:2
ID hilo interno:0
ID hilo interno:1
ID hilo interno:2
ID hilo interno:3

Agenda

74

I. *Modelo de programación sobre memoria compartida: Introducción*

II. *Posix Threads (Pthreads)*

I. *Gestión de hilos*

II. *Sincronización (Mutex | Variables Condición | Barreras)*

III. *Afinidad*

III. *OpenMP*

I. *Estructura de control paralela*

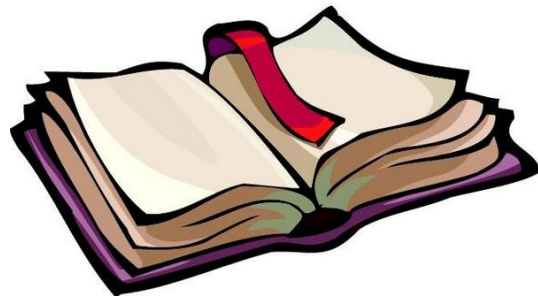
II. *Distribución de trabajo entre hilos*

III. *Gestión de ambiente de datos*

IV. *Sincronización*

V. *Funciones y variables de ambiente*

VI. *Afinidad*



OpenMP – Trabajo compartido

75

- OpenMP permite cuatro constructores para la distribución de trabajo entre hilos:
 - ▣ **do/parallel do, omp for:** distribuye las iteraciones de un for entre los hilos
 - ▣ **sections:** asigna bloques de código consecutivo e independiente a los hilos
 - ▣ **single:** especifica un bloque de código que será ejecutado por un único hilo. Existe una barrera implícita al final del bloque
 - ▣ **master:** similar a single. El bloque de código lo ejecuta sólo el Master thread. NO existe barrera al final del bloque

OpenMP – Trabajo compartido – Omp for

76

- El constructor **omp for**, por defecto distribuye las iteraciones del loop proporcionalmente entre los hilos.

```
int main(int argc, char* argv){
    int a[100];

    #pragma omp parallel {
        #pragma omp for
        for (int i = 0; i < 100; i++)
            a[i] = 2 * i;
    }
    return 0;
}
```

Simplificado

```
int main(int argc, char* argv){
    int a[100];

    #pragma omp parallel for
        for (int i = 0; i < 100; i++)
            a[i] = 2 * i;


    return 0;
}
```

OpenMP – Trabajo compartido – Omp for

77

- Error común y problema de anidamiento:

```
#pragma omp parallel for
for(int y=0; y<Y; ++y) {
    #pragma omp parallel for
    for(int x=0; x<X; ++x){
        f(x,y);
    }
}
```



El for más interno no se paraleliza!!!

Sólo se paraleliza el for externo.

El for interno corre en una secuencia como si el #pragma interno no existiese.

En la entrada de la directiva parallel interna OpenMP detecta que ya existe un grupo de hilos creados, y en vez de crear un nuevo grupo de T hilos, creará un grupo compuesto por sólo el hilo que llamó.

Solución

```
#pragma omp parallel for collapse(2) // 2 es el número de iteraciones anidadas
for(int y=0; y<Y; ++y) {
    for(int x=0; x<X; ++x){
        f(x,y);
    }
}
```

OpenMP – Trabajo compartido – Omp for

78

- Es importante entender como se distribuyen las iteraciones de un for entre los hilos.
- La distribución puede ser:
 - ▣ **Estática** (por defecto): se distribuyen proporcionalmente entre los hilos.
 - ▣ **Dinámica**: se distribuyen por demanda y de a cierta cantidad (**chunk**).
 - ▣ **Guiada**: distribución de iteraciones variable.
 - ▣ **Runtime**: distribución dada por la variable de entorno OMP_SCHEDULE
- Para indicar la distribución se debe agregar la cláusula **schedule**.

OpenMP – Trabajo compartido – Omp for

79

□ Distribución **estática**:

```
#pragma omp parallel for  
for(int y=0; y<8; ++y)  
    "Código"
```

```
#pragma omp parallel for schedule (static)  
for(int y=0; y<8; ++y)  
    "Código"
```

Si se ejecuta con dos hilos la distribución de iteraciones es:

Hilo 0 iteraciones $y=0, y=1, y=2, y=3$

Hilo 1 iteraciones $y=4, y=5, y=6, y=7$

OpenMP – Trabajo compartido – omp for

80

□ Distribución **dinámica**:

```
#pragma omp parallel for schedule (dynamic,2)
for(int y=0; y<4; ++y)
    "Código"
```

Cada hilo solicita iteraciones.

Si hay iteraciones se le asigna una cantidad dada por chunk (2 en el ejemplo).

Una distribución posible sería:

Hilo 0 iteraciones y=0, y=1

Hilo 1 iteraciones y=2, y=3

Hilo 1 iteraciones y=4, y=5

Hilo 0 iteraciones y=6, y=7

OpenMP – Trabajo compartido – omp for

81

- Distribución **guiada**:

```
#pragma omp parallel for schedule (guided,2)
    for(int y=0; y<4; ++y)
        "Código"
```

Cada hilo recibe dinámicamente bloques de iteraciones.

El bloque inicialmente es grande y va disminuyendo exponencialmente su tamaño hasta el valor especificado en chunk (2 en el ejemplo).

OpenMP – Trabajo compartido – Omp for

82

□ Distribución **runtime**:

```
#pragma omp parallel for schedule (runtime)
    for(int y=0; y<4; ++y)
        "Código"
```

La distribución de iteraciones se hace de acuerdo al valor de la variable de entorno OMP_SCHEDULE. Permite cambiar la política de planificación sin necesidad de compilar el código.

En Linux se define la variable OMP_SCHEDULE:

```
export OMP_SCHEDULE="static"
export OMP_SCHEDULE="dynamic,4"
export OMP_SCHEDULE="guided,2"
```

OpenMP – Trabajo compartido - Sections

83

- El constructor **sections**, se utiliza para distribuir secciones de código entre los hilos.

```
#pragma omp parallel
{
    #pragma omp sections {
        { Work1(); }
        #pragma omp section
        {Work2();
         Work3(); }
        #pragma omp section
        { Work4(); }
    }
}
```

Simplificado

```
#pragma omp parallel sections
{
    { Work1(); }
    #pragma omp section
    {Work2();
     Work3(); }
    #pragma omp section
    { Work4(); }
}
```

Work1, Work2-Work3 y Work4 son tres secciones que se se ejecutan en paralelo.
Cada sección se ejecuta una sola vez por un único hilo.
Work2 y Work3 deben ejecutarse en forma secuencial.

OpenMP – Trabajo compartido - Single

84

- El constructor **single**:

```
#pragma omp parallel
{
    Work1 ();
    #pragma omp single
    {
        Work2 ();
    }
    Work3;
}
```

Work1 y Work3 son tareas paralelas, todos los hilos la ejecutan.

Work2 es una tarea que ejecutará un único hilo, el primero que llegue.

Los demás hilos no podrán ejecutar Work3 hasta que termine la ejecución del hilo que ejecuta Work2.

OpenMP – Trabajo compartido - Master

85

- El constructor **master**:

```
#pragma omp parallel
{
    Work1 ();
    #pragma omp master
    {
        Work2 ();
    }
    Work3;
}
```

La diferencia con `single` es que `Work2` será ejecutado por el Master thread. Los demás hilos podrán seguir ejecutando `Work3`.

Agenda

86

I. *Modelo de programación sobre memoria compartida: Introducción*

II. *Posix Threads (Pthreads)*

I. *Gestión de hilos*

II. *Sincronización (Mutex | Variables Condición | Barreras)*

III. *Afinidad*

III. *OpenMP*

I. *Estructura de control paralela*

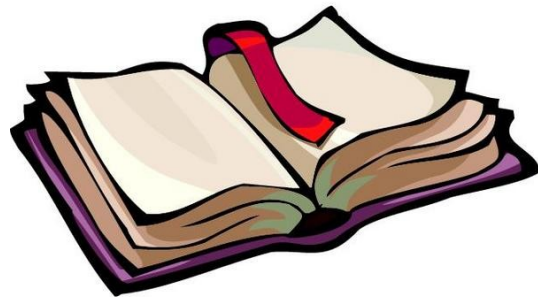
II. *Distribución de trabajo entre hilos*

III. *Gestión de ambiente de datos*

IV. *Sincronización*

V. *Funciones y variables de ambiente*

VI. *Afinidad*



OpenMP – Gestión de ambiente de datos

87

- En OpenMP las variables son visibles por todos los hilos.
- A veces es necesario declarar variables privadas para evitar condiciones de carrera.
- Existe una necesidad de intercambiar valores entre una región secuencial y una paralela.
- La gestión del ambiente de datos se realiza agregando clausulas a las directivas:
 - ▣ shared
 - ▣ private
 - ▣ firstprivate
 - ▣ lastprivate
 - ▣ reduction

OpenMP – Gestión de ambiente de datos

88

- **shared:** se comparten datos entre la región secuencial y la región paralela.
- Todos los hilos ven y acceden a los datos simultáneamente.
- Por defecto, todas las variables se comparten excepto las declaradas en las regiones paralelas.

```
int b=2;
#pragma omp parallel for shared(b)
    for(int a=0; a<50; ++a)
        printf("%d",a*b);
// a privada, b compartida
```

```
int b=2;
#pragma omp parallel for
    for(int a=0; a<50; ++a)
        printf("%d",a*b);
// a privada, b compartida
```


OpenMP – Gestión de ambiente de datos

89

- **private:** el dato en la región paralela es privado a cada hilo
- Cada hilo tiene una copia local privada la cual usa como variable temporal, **no se inicializa** y su valor no se mantiene fuera de la región paralela
- Por defecto, el índice del for que sigue a un `#pragma omp for` y las variables definidas dentro de la región paralela son privados.

```
int b=2;
int j,a;
#pragma omp parallel for private(j,a)
    for(a=0; a<50; ++a)
        j=a*b;
// j,a privadas, b compartida
```

```
int b=2;
int j;
int a;
#pragma omp parallel for private(j)
    for(a=0; a<50; ++a){
        int c=a*b*j;
        j=a*b;
    }
// j,a,c privadas, b compartida
```

OpenMP – Gestión de ambiente de datos

90

- Ejemplo de ejecución con **private**:

```
int j = 4;  
#pragma omp parallel for private(j)  
    for(a=0; a<3; a++){  
        j+=a;  
        printf("%d\n",j);  
    }  
  
printf("Final %d\n",j);
```

Posible salida:

4321

0

543243

1234

Final 4

OpenMP – Gestión de ambiente de datos

91

- **firstprivate:** idem private pero el valor se inicializa con el valor original de la variable.

```
int j = 4;
#pragma omp parallel for firstprivate(j)
    for(a=0; a<3; a++){
        j+=a;
        printf("%d\n", j);
    }

printf("Final %d\n", j);
```

Posible salida:

4

6

7

5

Final 4

OpenMP – Gestión de ambiente de datos

92

- **lastprivate:** idem private pero el valor finaliza con el ultimo valor calculado.

```
int j;  
#pragma omp parallel for lastprivate(j)  
    for(a=0; a<3; a++){  
        j=a;  
        printf("%d\n",j);  
    }  
  
printf("Final %d\n",j);
```

Posible salida:

3

2

0

1

Final 1

OpenMP – Gestión de ambiente de datos

93

- **reduction:** une el resultado de todos los hilos mediante alguna operación.

```
int sum = 0;
#pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < count; ++i)
        sum += i;
printf("%d",sum);
```

```
int factorial(int number) {
    int fac = 1;
    #pragma omp parallel for reduction(*:fac)
        for(int n=2; n<=number; ++n)
            fac *= n;
    return fac;
}
```

Posibles operandos:

+

-

*

&

|

^

&&

||

Max

Min

Agenda

I. *Modelo de programación sobre memoria compartida: Introducción*

II. *Posix Threads (Pthreads)*

I. *Gestión de hilos*

II. *Sincronización (Mutex | Variables Condición | Barreras)*

III. *Afinidad*

III. *OpenMP*

I. *Estructura de control paralela*

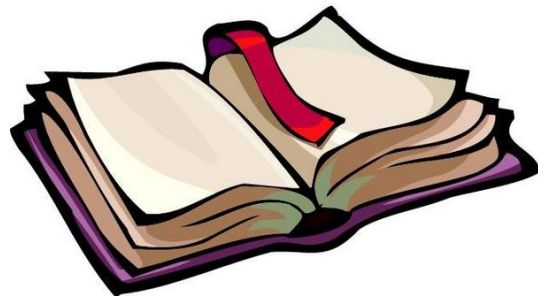
II. *Distribución de trabajo entre hilos*

III. *Gestión de ambiente de datos*

IV. *Sincronización*

V. *Funciones y variables de ambiente*

VI. *Afinidad*



OpenMP – Sincronización

95

- Para realizar sincronización OpenMP incluye las cláusulas:
 - ▣ `critical`
 - ▣ `atomic`
 - ▣ `ordered`
 - ▣ `barrier`
 - ▣ `nowait`

- Estas cláusulas pueden usarse en conjunto con las directivas.

OpenMP – Sincronización

96

- **critical:** encierra un bloque de código (región crítica) que debe ser ejecutado por un hilo a la vez.

```
int sum = 0, prod = 1;  
#pragma omp parallel for  
for (int i = 1; i < count; i++){  
    sum += i;  
    prod *=i;  
}
```

A large red question mark is centered within a black rectangular box, which is positioned to the right of the code block. This likely represents a question or a point of discussion regarding the code's execution or synchronization.

OpenMP – Sincronización

97

```
int sum = 0, prod = 1;
#pragma omp parallel for
for (int i = 1; i < count; i++){
    sum += i;
    prod *= i;
}
```

Las variables sum y prod son shared por defecto.

Pero “sum += i” y “prod *= i” no son instrucciones atómicas.

Condición de carrera!!!

Solución

```
int sum = 0, prod = 1;
#pragma omp parallel for
for (int i = 1; i < count; i++){
    #pragma omp critical (lock1) { //lock1:nombre de la región crítica
        sum += i;
        prod *= i;
    }
}
```

OpenMP – Sincronización

98

- **atomic:** similar a critical pero encierra una sola instrucción

```
int sum = 0;
#pragma omp parallel for
for (int i = 0; i < count; i++)
    #pragma omp atomic
    sum += i;
```

OpenMP – Sincronización

99

- **ordered:** se utiliza cuando es necesario que las instrucciones se ejecuten según el orden de las iteraciones. El orden de la ejecución paralela sería equivalente a una ejecución secuencial.

```
#pragma omp parallel for  
    for (int i = 0; i < 3; i++)  
        printf("%d\n",i);
```

Posible salida:

0
3
2
1

```
#pragma omp parallel for ordered  
    for (int i = 0; i < 3; i++)  
        printf("%d\n",i);
```

Posible salida (ordered):

0
1
2
3

OpenMP – Sincronización

100

- **barrier:** los hilos esperan en un punto en el código para continuar la ejecución.

```
#pragma omp parallel {  
    "Codigo"      //Todos los hilos ejecutan esto  
  
    #pragma omp barrier // Se esperan en este punto  
  
    "Otro código" //Cuando todos alcanzaron la barrera continuan la ejecución  
}
```

OpenMP – Sincronización

101

- **nowait:** especifica que los hilos que terminaron su trabajo puedan continuar sin esperar al resto

```
#pragma omp parallel{  
  #pragma omp for nowait  
  for(int n=0; n<10; ++n){  
    "Código del loop"  
  }  
  // Esta línea puede ser alcanzada mientras otros hilos aún están en el loop.  
  "Código fuera del loop"  
}
```

Agenda

102

I. *Modelo de programación sobre memoria compartida: Introducción*

II. *Posix Threads (Pthreads)*

I. *Gestión de hilos*

II. *Sincronización (Mutex | Variables Condición | Barreras)*

III. *Afinidad*

III. **OpenMP**

I. *Estructura de control paralela*

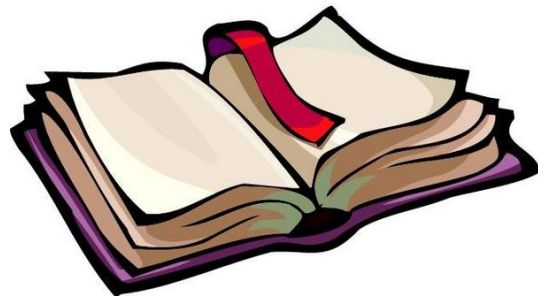
II. *Distribución de trabajo entre hilos*

III. *Gestión de ambiente de datos*

IV. *Sincronización*

V. **Funciones y variables de ambiente**

VI. *Afinidad*



OpenMP – Funciones y variables de ambiente

103

- OpenMP posee un conjunto de funciones y variables de ambiente que permiten configurar u obtener valores para la ejecución.
- Funciones:
 - ▣ `omp_get_thread_num()`: retorna el identificador del hilo
 - ▣ `omp_set_num_threads(int num_threads)`: determina la cantidad de hilos que se crearán
 - ▣ `omp_get_num_threads()`: retorna cual es la cantidad de hilos que se crearon
 - ▣ `omp_set_nested(int value)`: habilita o deshabilita el anidamiento de hilos
- Variables:
 - ▣ `OMP_NUM_THREADS`: idem función `omp_set_num_threads`
 - ▣ `OMP_SCHEDULE` = [static, dynamic]
 - ▣ `OMP_NESTED`: idem función `omp_set_nested`

Agenda

104

I. *Modelo de programación sobre memoria compartida: Introducción*

II. *Posix Threads (Pthreads)*

I. *Gestión de hilos*

II. *Sincronización (Mutex | Variables Condición | Barreras)*

III. *Afinidad*

III. *OpenMP*

I. *Estructura de control paralela*

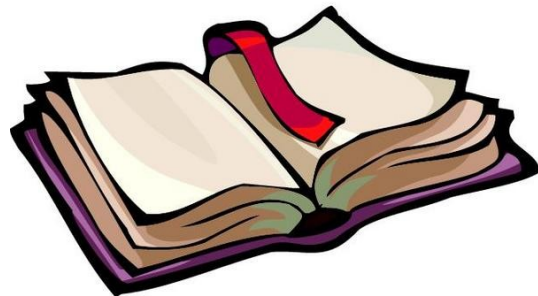
II. *Distribución de trabajo entre hilos*

III. *Gestión de ambiente de datos*

IV. *Sincronización*

V. *Funciones y variables de ambiente*

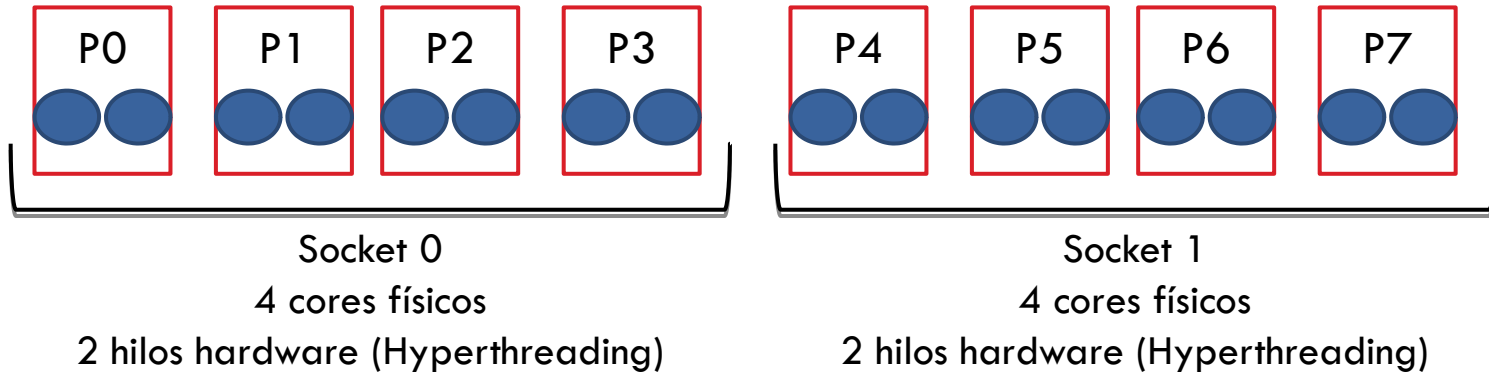
VI. *Afinidad*



OpenMP – Afinidad

105

- OpenMP permite controlar la afinidad mediante la cláusula **proc_bind** o mediante las variables de ambiente **OMP_PLACES** y **OMP_PROC_BIND**.
- Suponer la siguiente arquitectura:



OpenMP considera esta arquitectura como 8 sitios para ubicar hilos
P0,P1,P2,P3,P4,P5,P6,P7

OpenMP – Afinidad

106

- Mediante la variable de ambiente OMP_PLACES se pueden ubicar los hilos por thread ID en los lugares que se desee:

OMP_PLACES="{0,1}, {2,3}, {4,5}, {6,7}, {8,9}, {10,11}, {12,13}, {14,15}"

- Esto es equivalente a:

- **OMP_PLACES="{0:2}, {2:2}, {4:2}, {6:2}, {8:2}, {10:2}, {12:2}, {14:2}"**

{l:L} significa: a partir del thread ID l ubicar L hilos

- **OMP_PLACES={0:2}:8:2**

A partir del thread ID 0 ubicar dos hilos en 8 sitios de 2 lugares cada uno

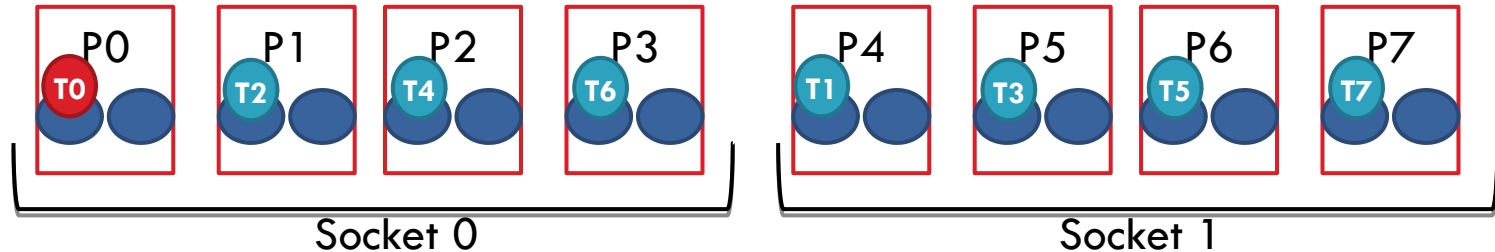
OpenMP – Afinidad

107

- Con la variable `OMP_PLACES` también se pueden ubicar alternadamente por sockets:

`OMP_PLACES=sockets`

Cada sitio se corresponde a un socket individual



OpenMP – Afinidad

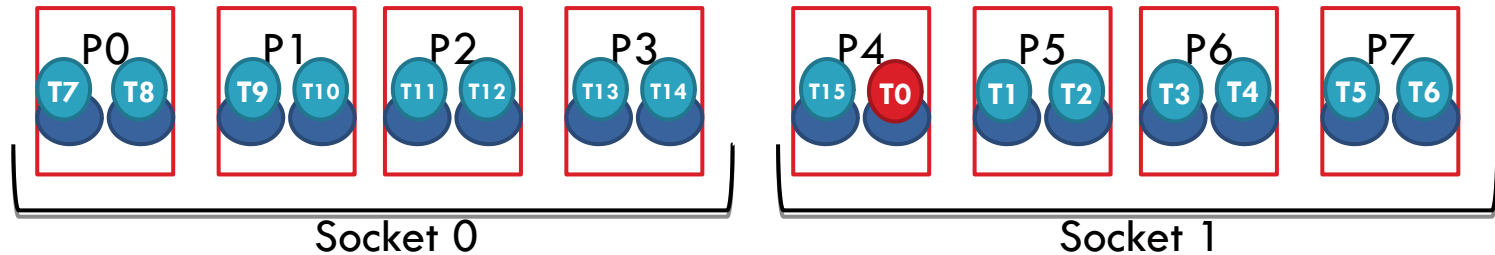
108

- O se pueden ubicar consecutivamente por thread ID a partir de la ubicación del master thread (T0)

OMP_PLACES=threads

Cada sitio se corresponde a un hilo-hardware individual (Hilos de HT)

Si el master thread está en el sitio P4 en el segundo hilo-hardware



OpenMP – Afinidad

109

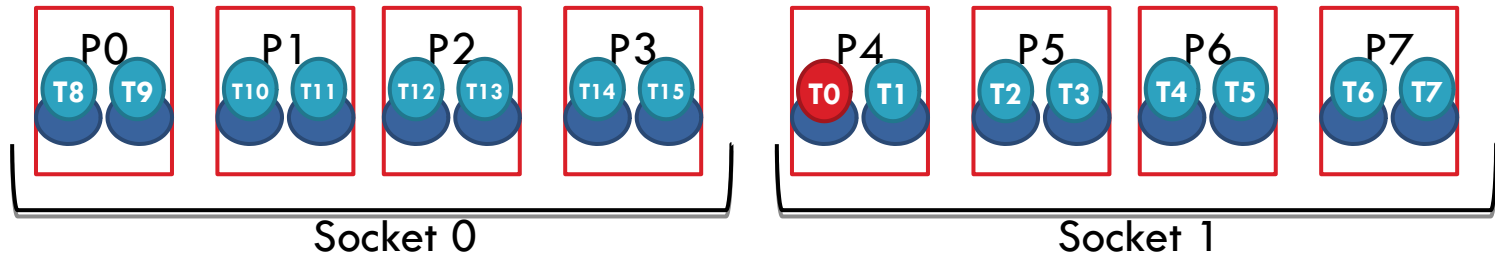
- Es posible además ubicarlos de acuerdo a los cores físicos:

OMP_PLACES=cores

Cada sitio se corresponde a un core individual (que puede tener 2 hilos-hardware HT)

Ahora el master thread está en el sitio P4 NO TIENE EN CUENTA el hilo-hardware

Comienza desde el primero



OpenMP – Afinidad

110

- Utilizando **proc_bind** o la variable de ambiente **OMP_PROC_BIND** (en combinación con **OMP_PLACES=cores**) se pueden obtener tres formas más de distribución:
 - ▣ **Close**
 - ▣ **Spread**
 - ▣ **Master**

OpenMP – Afinidad

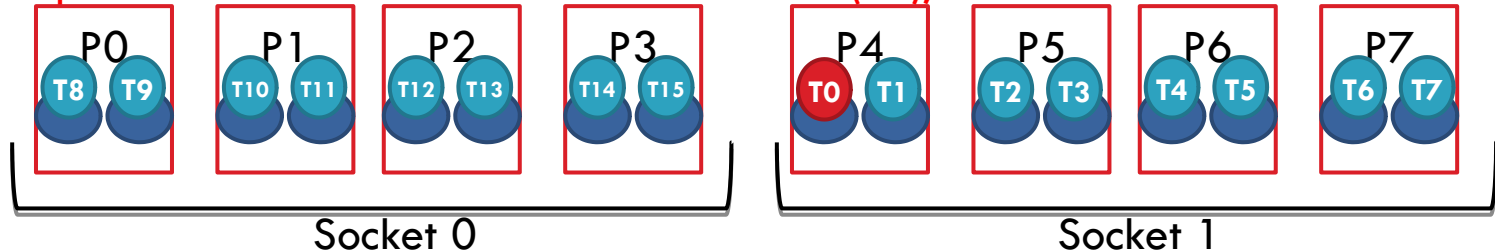
111

- Política de afinidad **close**: ubica los hilos cerca de manera que puedan compartir datos en cache.
- Se puede obtener con la cláusula **proc_bind** o con la variable **OMP_PROC_BIND**:

```
#pragma omp parallel proc_bind(close)
{
    ...
}
```

OMP_PROC_BIND=close

Depende de la ubicación del master thread (T0), si está en el sitio 4:



OpenMP – Afinidad

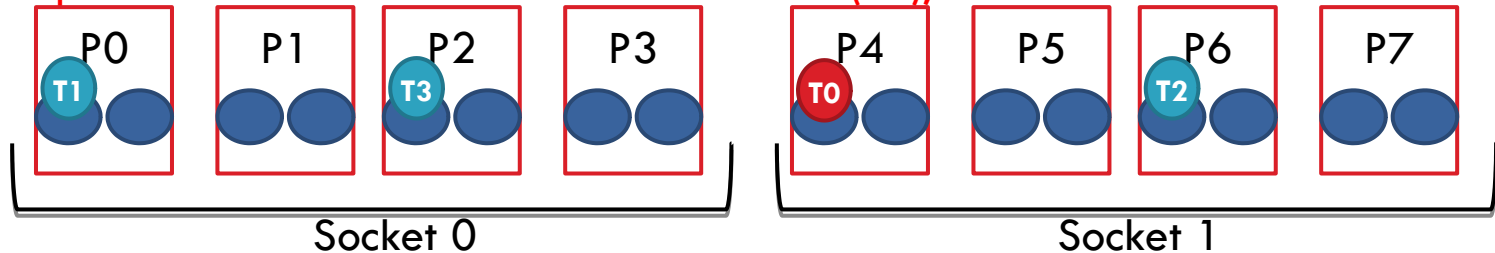
112

- Política de afinidad **spread**: ubica los hilos dispersos de manera que **NO** compartan datos en cache. **Se utiliza cuando se corren menos hilos que número de cores disponibles.**
- Se puede obtener con la cláusula **proc_bind** o con la variable **OMP_PROC_BIND**:

```
#pragma omp parallel proc_bind(spread)
{
    ...
}
```

OMP_PROC_BIND=spread

Depende de la ubicación del master thread (T0), si está en el sitio 4:



OpenMP – Afinidad

113

- Política de afinidad **master**: ubica los hilos cercanos al master thread (T0).
- Se puede obtener con la cláusula **proc_bind** o con la variable **OMP_PROC_BIND**:

```
#pragma omp parallel proc_bind(master)
{
    ...
}
```

OMP_PROC_BIND=master

Depende de la ubicación del master thread (T0), si está en el sitio 4 y sólo hay 4 hilos:

