AC 21/22 - Resumen seminarios prácticas

Seminario 0 - Entorno de programación: ATCgrid y gestor de carga de trabajo

1. Cluster de prácticas (atcgrid)

Componentes

- Nodos de cómputo, tres servidores ATCgrid1 a 3
  - Dos procesadores (o sea, dos socket). Cada procesador tiene 6 núcleos, con dos niveles de caché L1 y L2.
  - Los procesadores son multithread, así que cada procesador tiene 6 cores y 12 hebras
  - En total, 24 cores lógicos y 12 cores físicos
- Nodo de cómputo, un servidor ATCgrid 4, más tocho que los otros 3
  - 2 procesadores, con 16 cores cada uno y multithreading, así que 32 hebras en cada uno

    Entonces, cores físicos: 32, cores lógicos: 64
- Nodo front-end, desde donde mandamos las cosas a los otros 4 nodos

Acceso

Cada usuario tiene un home en el nodo front-end del clúster ATCgrid.

Para ejecutar comandos (srun, sbatch, squeue...), con un cliente ssh (secure shell):

```
Linux: $ ssh -X username@atcgrid.ugr.es (pide password del usuario “username”)
```

Para cargar y descargar ficheros (put hello, get slurm-9.out, ...), con un cliente sftp (secure file transfer protocol)

```
Linux: $ sftp username@atcgrid.ugr.es (pide password del usuario “username”)
```

2. Gestor de carga de trabajo (workload manager)

Se ejecutará en el front-end con conexión ssh:

Ejemplo	Explicación
`srun -pac -Aac ./hello` `srun -p ac4 -A ac lscpu`	`srun` envía a ejecutar un trabajo (en los ejemplos, el ejecutable hello, y lscpu) a través de una cola slurm. Si aparece `-p`, se envía a nodos de la cola especificada con `-p` (un trabajo solo puede usar un nodo de la cola `ac`).

`sbatch -p ac script.sh`
`sbatch -p ac --wrap "echo Hola"`
`sbatch -p ac --wrap "./hello"`
`sbatch -p ac --wrap "echo Hola ; ./hello"`
	`sbatch` envía a ejecutar un *script* (en este caso script.sh, "echo Hola", "./hello" y "echo Hola ; ./hello") a través de una cola slurm. La salida se devuelve en un fichero. La ejecución con `srun` es *interactiva*, con `sbatch` es en *segundo plano*. Se recomienda usar `sbatch`

squeue	Muestra todos los trabajos en ejecución y los que están encolados
scancel jobid	Elimina el trabajo con identificador "jobid"
sinfo	Lista información de las particiones (colas) y de los nodos
sinfo -p ac -o"%10D %10G %20b %f"	Lista los nodos (D), los recursos (G), y las características activas (b) y disponibles (f) en la partición especificada (-p) (%[[.]size]type[suffix])

Ejemplo	Explicación
`sbatch -pac -n1 -c12 script.sh` `srun -pac -n1 -c12 helloomp`	Slurm está configurado para asignar recursos a los procesos (llamados tasks en slurm) a nivel de core físico. Esto significa que por defecto slurm asigna un core a un proceso, para asignar `x` se debe usar con `sbatch/srun` la opción `--cpus-per-task=x` (o su forma abreviada `-cx`). Para asegurar que solo se crea un proceso hay que incluir `--ntasks=1` (`-n1`) en `sbatch/srun`.
`sbatch -pac -n1 -c12 --hint=nomultithread script.sh` `srun -pac -n1 -c12 --hint=nomultithread helloomp`	En slurm, por defecto, cpu se refiere a cores lógicos (ej. en la opción `-c`), si no se quieren usar cores lógicos hay que añadir la opción `--hint=nomultithread` a `sbatch/srun`. `sbatch` tendrá en cuenta `--hint=nomultithread` si se usa `srun` dentro del script delante del ejecutable.
`sbatch -pac -n1 -c12 --exclusive --hint=nomultithread script.sh` `srun -pac -n1 -c12 --exclusive --hint=nomultithread helloomp`	Para que no se ejecute más de un proceso en un nodo de cómputo de atcgrid hay que usar `--exclusive` con `sbatch/srun` (se recomienda no utilizarlo en los srun dentro de un script).

3. Ejemplo de script

script_helloomp.sh

```
#!/bin/bash
#Órdenes para el Gestor de carga de trabajo (no intercalar instrucciones del script):
#1. Asigna al trabajo un nombre
#SBATCH --job-name=helloOMP
#2. Asignar el trabajo a una cola (partición)
#SBATCH --partition=ac
#3. Asignar el trabajo a un account
#SBATCH --account=ac
#4. Para que el trabajo no comparta recursos #SBATCH -exclusive
#5. Para que se genere un único proceso del SO que pueda usar un máximo de 12 núcleos
#SBATCH --ntasks 1 --cpus-per-task 12

#Obtener información de las variables del entorno del Gestor de carga de trabajo:
echo "Id. usuario del trabajo: $SLURM_JOB_USER"
echo "Id. del trabajo: $SLURM_JOBID"
echo "Nombre del trabajo especificado por usuario: $SLURM_JOB_NAME"
echo "Directorio de trabajo (en el que se ejecuta el script): $SLURM_SUBMIT_DIR"
echo "Cola: $SLURM_JOB_PARTITION"
echo "Nodo que ejecuta este trabajo:$SLURM_SUBMIT_HOST"
echo "Nº de nodos asignados al trabajo: $SLURM_JOB_NUM_NODES"
echo "Nodos asignados al trabajo: $SLURM_JOB_NODELIST"
echo "Nº CPUs disponibles para el trabajo en el nodo: $SLURM_JOB_CPUS_PER_NODE"

#Instrucciones del script para ejecutar código:
echo -e "\n 1. Ejecución helloOMP una vez sin cambiar nº de threads (valor por defecto):\n"
srun ./HelloOMP
echo -e "\n 2. Ejecución helloOMP varias veces con distinto nº de threads:\n"
for ((P=12;P>0;P=P/2))
do
   #export OMP_NUM_THREADS=$P
   echo -e "\n - Para $P threads:"
   srun --cpus-per-task=$P --hint nomultithread ./HelloOMP
done
```

Órdenes para Gestor de carga de trabajo

Para imprimir variables del Gestor de carga de trabajo

Instrucciones del script

No olvidar poner **srun** delante del ejecutable
Se puede descomentar `export OMP_NUM_THREADS=$P` y quitar `--cpus-per-task=$P`

BP0 -- Apuntes

- Cores físicos: num procesadores * num cores
- Cores lógicos: num procesadores * num cores * num hebras

Seminario 1 - Herramientas de programación paralela I: directivas OpenMP

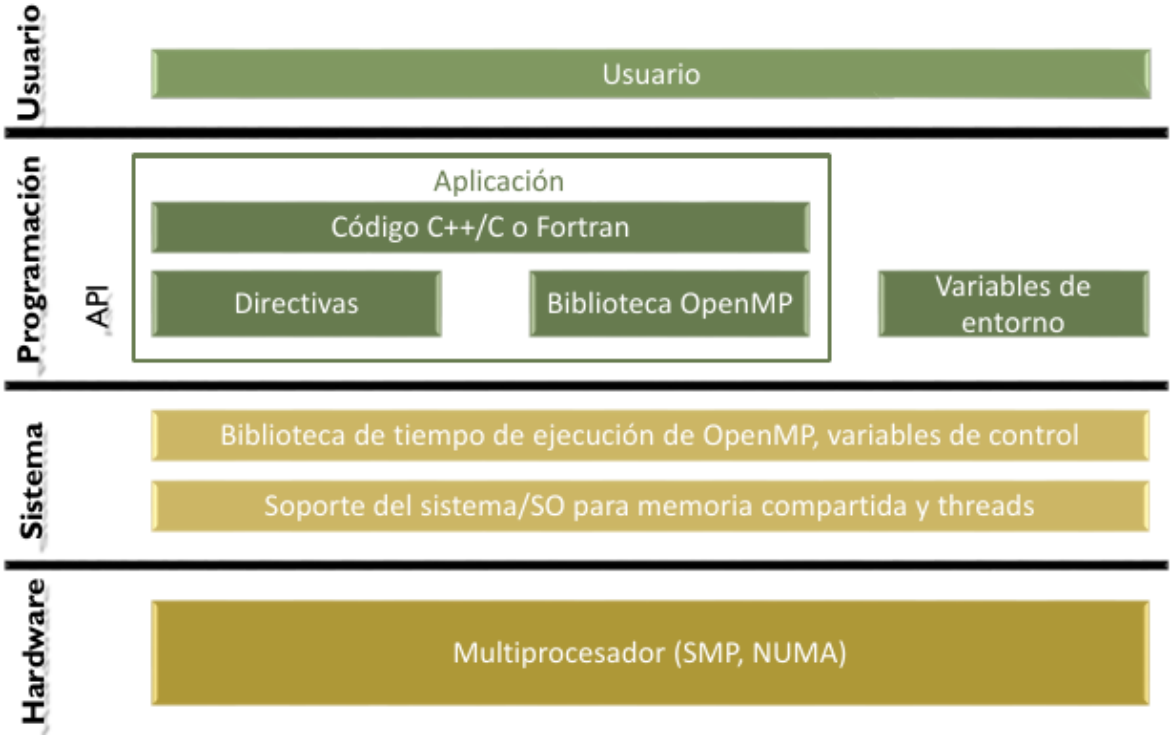
1. OpenMP

Open Multi-Processing:

- Especificaciones abiertas (Open) para multiprocesamiento (Multi-processing) generadas mediante trabajo colaborativo de diferentes entidades interesadas de la industria del hardware y software, además del mundo académico y del gobierno (Intel, IBM, Oracle, Microsoft, NASA, universidades...)

Es una API para C/C++ y Fortran para escribir código paralelo , usando directivas y funciones, con el paradigma/estilo de programación de variables compartidas para ejecutar aplicaciones en paralelo en varios threads.

- **API (Application Programming Interface):** Capa de abstracción que permite al programador acceder cómodamente a través de una interfaz a un conjunto de funcionalidades. La API OpenMP define/comprende:
  - Directivas del compilador
  - Funciones de biblioteca
  - Variables de entorno



OpenMP es una herramienta de programación paralela...

- No automática (no extrae paralelismo implícito, es tarea del programador)
- Con un modelo de programación...

    - Basado en el paradigma de variables compartidas
    - Multithread
    - Basado en directivas del compilador y funciones (el código paralelo OpenMP es código escrito en un lenguaje secuencial + directivas y func de OpenMP)
- Portable (API específica para C/C++ y Fortran, la mayor parte de las prataformas SO/hardware tienen implementaciones OpenMP)
- Prácticamente estándar (adoptada y desarrollada por los mayores vendedores de hardware y software)

2. Componentes de OpenMP

- **Directivas**: el preprocesador del compilador las sustituye por código
- **Funciones**: fijar parámetros, obtener parámetros...
- **Variables de entorno**: para fijar parámetros previa ejecución

Sintaxis directivas C/C++

#pragma omp	[nombre]	OPC[clausula, [...] ...]	\n
Obligatorio	Obligatorio	Opcional. Pueden aparecer en cualquier orden	Salto de línea obligatorio. La directiva engloba al bloque estructurado

- El nombre define y controla la acción que se realiza
    - Ej.: parallel, for, section ...
- Las cláusulas especifican adicionalmente la acción o comportamiento, la ajustan
    - Ej.: private, schedule, reduction ...
- Las comas separando cláusulas son opcionales
- Se distingue entre mayúsculas y minúsculas

```
EJEMPLO:
#pragma omp parallel num_threads(8) if(N>20)
```

Portabilidad

- Compilación C/C++: `gcc -fopenmp` / `g++ -fopenmp`
- Directivas: las directivas no se tendrán en cuenta si no se compila usando OpenMP
    - -fopenmp, -openmp, etc.
- Funciones: se evitan usando compilación condicional.
    - Para C/C++: usando _OPENMP y #ifdef ... #endif
    - _OPENMP se define cuando se compila usando OpenMP

Definiciones

- Directiva ejecutable (executable directive):
    - Aparece en código ejecutable
- Bloque estructurado (structured block):

    - Un conjunto de sentencias con un única entrada al principio del mismo y una única salida al final.
    - No tiene saltos para entrar o salir
    - Se permite exit() en C/C++
- Construcción (construct) (extensión estática o léxica):
    - Directiva ejecutable + [sentencia, bucle o bloque estructurado]
- Región (extensión dinámica):
    - Código encontrado en una instancia concreta de la ejecución de una construcción o subrutina de la biblioteca OpenMP
    - Una construcción puede originar varias regiones durante la ejecución
    - Incluye: código de subrutinas y código implícito introducido por OpenMP

3. Directivas

La directiva define la acción que se realiza. En color, las directivas que vamos a estudiar.

Subrayadas, las directivas con barrera implícita al final.

DIRECTIVA	ejecutable	declarativa	
con bloque estructurado	parallel sections, worksharing, single master critical ordered		Con sentencias
bucle	DO/for		
simple (una sentencia)	atomic		
autónoma (sin código asocidado)	barrier, flush	threadprivate	sin

Directiva parallel

```
#pragma omp parallel [clause[[,]clause]...]
    //bloque estructurado
```

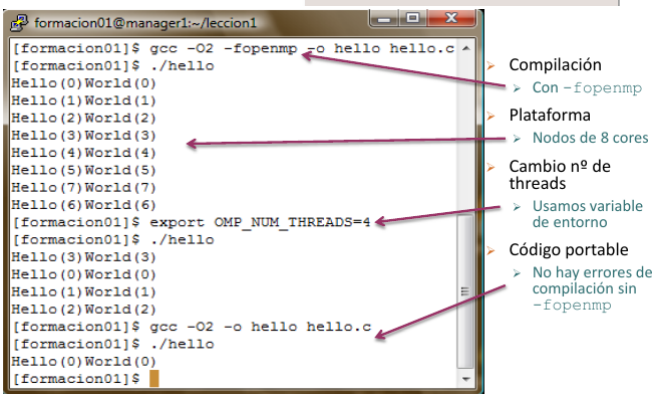
- Especifica qué cálculos se ejecutarán en paralelo
- Un thread (master) crea un conjunto de threads cuando alcanza una Directiva parallel
- Cada thread ejecuta el código incluido en la región que conforma el bloque estructurado
- No reparte tareas entre threads
- Barrera implícita al final
- Se pueden usar de forma anidada

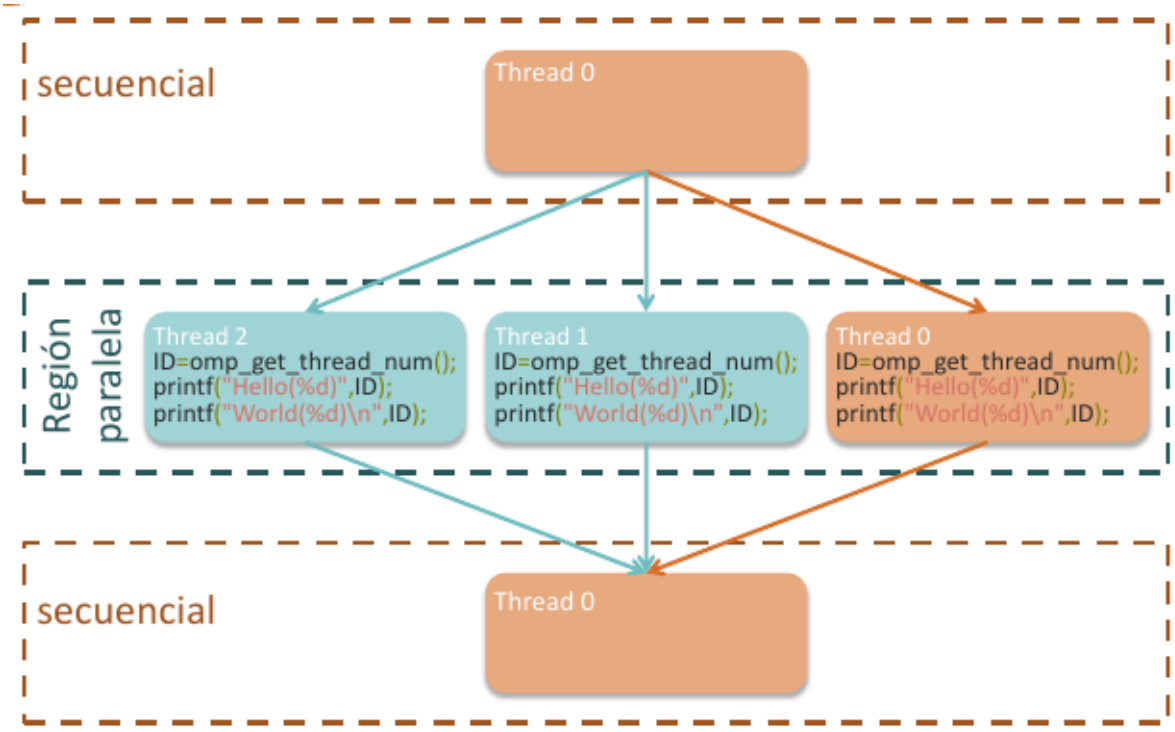
Ejemplo: hello world

Imprime en pantalla con dos `printf` distintos “Hello World” y el identificador del thread que imprime

hello.c

```
#include <stdio.h>

#ifdef _OPENMP
   #include <omp.h>
#else
   #define omp_get_thread_num() 0
#endif

main() {

 int ID;
 #pragma omp parallel private(ID)
 {
   ID = omp_get_thread_num();
   printf("Hello(%d)",ID);
   printf("World(%d)\n",ID);
 }

}
```





¿Cómo se enumeran y cuántas threads se usan?

Se enumeran comenzando desde 0 (0… nº threads-1). El master es la 0

¿Cuántos thread se usan en las ejecuciones anteriores? Empezando por las de más prioridad, a menos:

- El nº fijado por el usuario modificando la variable de entorno OMP_NUM_THREADS
  - Con el shell o intérprete de comandos Unix csh (C shell): `setenv OMP_NUM_THREADS 4`
  - Con el shell o intérprete de comandos Unix ksh (Korn shell) o bash (Bourne-again shell): `export OMP_NUM_THREADS=4`
- Fijado por defecto por la implementación: normalmente el nº de cpu de un nodo, aunque puede variar dinámicamente

Directivas de trabajo compartido (worksharing)

- `#pragma omp for` - para distribuir las iteraciones de un bucle entre threads (paralelismo de datos)

  ```
  #pragma omp for [clause[[,]clause]...]
      //for-loop
  ```

  Se tiene que conocer el nº de iteraciones. No se puede bucles do-while. Las iteraciones deben poder ser paralelizables (la herramienta no sabe). La asignación se hace sola a no ser que usemos cláusula schedule.

  - Tipo de paralelismo: de datos o a nivel de bucle
  - Tipo de estructuras de procesos/tareas: descomposición del dominio, divide y vencerás
  - Sincronización: barrera implícita al final, no al principio
  - Asignación de tareas: lo hace la herramienta, a no ser que usemos cláusula schedule
- `#pragma omp sections` - para distribuir trozos de código independientes entre las threads (paralelismo de tareas)

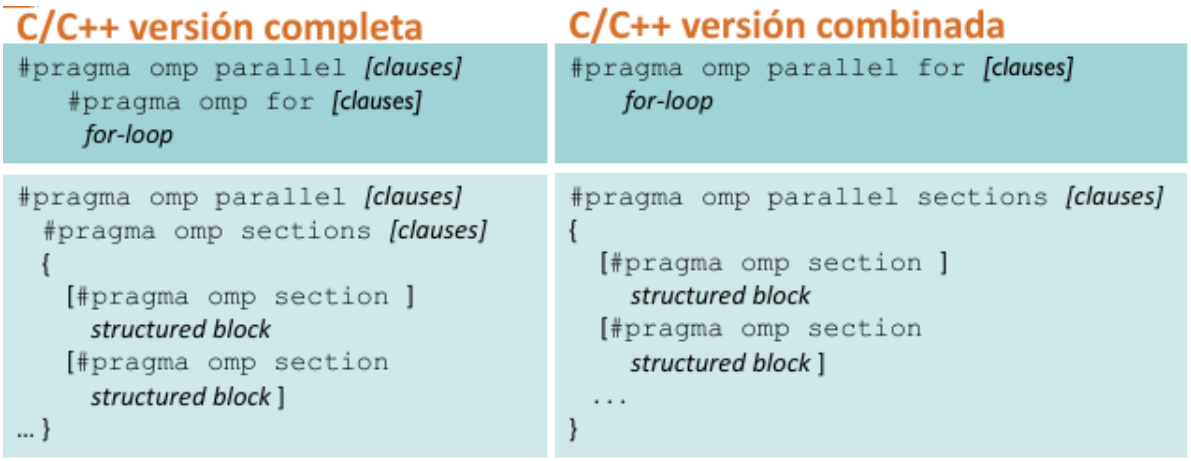
  ```
  #pragma omp sections [clause[[,]clause]...] {
      [#pragma omp section ]
          //structured block
      [#pragma omp section ]
          //structured block
          ...
  }
  ```

  - Tipo de paralelismo: de tareas o a nivel de función
  - Tipo de estructuras de procesos/tareas: maestro-esclavo, cliente-servidor, flujo de datos, descomposición del dominio, divide y vencerás
  - Sincronización: barrera implícita al final, no al principio
  - Asignación de tareas: lo hace la herramienta
- `#pragma omp single` - para que uno de los threads ejecute un trozo de código secuencial, útil cuando algo no es thread-safe como la E/S

  ```
  #pragma omp single [clause[[,]clause]...] {
      //structured block
  ```

  - Tipo de paralelismo: no es paralelismo, ejecuta secuencialmente un trozo
  - Sincronización: barrera implícita al final, no al principio
  - Asignación de tareas: lo hace la herramienta, puede ser cualquier thread

Combinar parallel con worksharing



Directivas básicas de comunicación y sincronización

- `#pragma omp barrier` - para definir una barrera, un punto en el que las hebras se esperan entre sí. Al final de parallel y de las worksharing hay barrera implícita
- `#pragma omp critical` - evita que varios thread accedan a variables compartidas a la vez, evita condición de carrera

  ```
  #pragma omp critical [(name)]
      //bloque estructurado
  ```

  "Name" permite evitar cerrojos. Sección crítica es el código que accede a variables compartidas
- `#pragma omp atomic` - puede ser una alternativa a critical más eficiente, para operaciones atómicas

  ```
  #pragma omp atomic
      x <binop> = expre. … [+, *, -, /, ^, &, |, <<, >>]

  #pragma omp atomic
      x ++, ++x, x-- o -- x.
  ```

Directiva master

```
#pragma omp master
    //structured block
```

No es una directiva de trabajo compartido, no tiene barrera implícita.

BP1 -- Apuntes

- Acordarse siempre que usemos master de poner una barrier al final

Seminario 2 - Herramientas de programación paralela II: cláusulas OpenMP

1. Cláusulas

Las cláusulas ajustan el comportamiento de las directivas. Las directivas con cláusulas son:

- parallel
- worksharing: for, sections, single, workshare
- parallel for, parallel sections
    - aceptan cláusulas de las dos directivas excepto nowait

No aceptan cláusulas:

- master
- sincronización/consistencia: critical, barrier, atomic, flush
- ordered
- threadprivate

DIRECTIVA	ejecutable	declarativa	
con bloque estructurado	*parallel* *sections, worksharing, single* master critical ordered		Con sentencias
bucle	*DO/for*		
simple (una sentencia)	atomic		
autónoma (sin código asocidado)	barrier, flush	threadprivate	sin

Las que aceptan cláusulas están en cursiva.

2. Ámbito de los datos por defecto. Cláusulas de compartición de datos

En color, las cláusulas que vamos a comentar

TIPO	Cláusula	Directivas					
		parallel	DO/for	sections	single	parallel DO/for	parallel sections
Control nº threads	If (1)	X				X	X
	num_threads (1)	X				X	X
Control ámbito de las variables	shared	X				X	X
	private	X	X	X	X	X	X
	lastprivate		X	X		X	X
	firstprivate	X	X	X	X	X	X
	default (1)	X				X	X
	reduction	X	X	X		X	X
Copia de valores	copyin	X				X	X
	copyprivate				X		
Planifica. iteraciones bucle	schedule (1)		X			X	
	ordered (1)		X			X	
No espera	nowait		X	X	X		

Ámbito de los datos o atributos de compartición

Como regla general para regiones paralelas: las variables declaradas fuera de una región y las dinámicas son compartidas por las threads de la región. Las variables declaradas dentro son privadas.

Son excepciones el índice de los bucle for (predeterminado ámbito privado) y variables declaradas static.

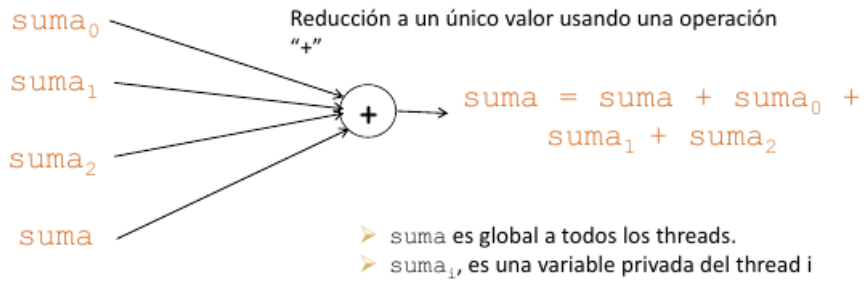
Cláusulas de compartición de datos

- shared(list) - se comparten las variables de la lista list por todas las threads. Precaución cuando al menos un thread lee lo que otro escribe en alguna variable de la lista.
- private(list) - el valor de entrada y de salida está indefinido, aunque la variable esté declarada fuera de la construcción.
- lastprivate(lista) - la acción de private, y la copia (al salir de la región paralela) del último valor (en una ej. secuencial) de las variables de la lista se hace pública.
    - En un bucle, el valor de la última iteración
    - En un sections, el valor tras la última sección
- firstprivate(lista) - la acción de private, e inicializa las variables de la lista al entrar en la región paralela.
- default(none/shared) - sólo puede haber una cláusula default
    - Con none, el programador debe especificar el alcance de todas las variables usadas en la construcción, menos variables threadprivate e índices de bucles for.
    - Con default, se ponen a su ámbito por defecto
        - Se pueden excluir del ámbito por defecto usando shared, private, firstprivate, lastprivate, reduction...

3. Cláusulas de comunicación/sincronización

- reduction(operator:list) - reduce al final los valores de list operando con el operador suministrado. Comunicación colectiva todos-a-uno.

C/C++	
tipo	**Valor inicial variables locales**
+	0
-	0
*	1
&	~0 (bits a 1)
|	0
^	0
&&	1
||	0



- copyprivate(list) - solo se puede usar en single. Permite que una variable privada de una hebra se copia a las variables privadas del mismo nombre del resto de threads (difusión). Útil para entrada de variables.

```
#pragma omp parallel
{  int a;
   #pragma omp single copyprivate(a)
   {
     printf("\nIntroduce valor de inicialización a: ");
     scanf("%d", &a );
     printf("\nSingle ejecutada por el thread %d\n",
         omp_get_thread_num());
   }
   #pragma omp for
   for (i=0; i<n; i++)  b[i] = a;
}
```

Seminario 3 - Herramientas de programación paralela III: interacción con el entorno en OpenMP y evaluación de prestaciones

Objetivos: consulta y modificación de parámetros (número de threads, tipo de planificación de tareas...).

Relaicón con el entorno de ejecución, de más a menos prioridad...

- Variables de control internas
- Variables de entorno
- Funciones del entorno de ejecución
- Cláusulas (no modifican variables de control)

1. Variables de control

Variables de control internas que afectan a parallel

Variable de control	Ámbito	Valor (valor inicial)	¿Qué controla?	Consultar /Modificar
dyn-var	entorno de datos	true/false (depende de la implementación)	Ajuste dinámico del nº de threads	sí(f) /sí(ve,f)
nthreads-var	entorno de datos	número (depende de la implementación)	threads en la siguiente ejecución paralela	sí(f) /sí(ve,f)
thread-limit-var	entorno de datos	número (depende de la implementación)	Máximo nº de threads para todo el programa	sí(f) /sí(ve,-)
nest-var		true/false (false)	Paralelismo anidado	sí(f) /sí(ve,f)

V3.0 en gris f: función, ve: variable de entorno nest-var: en desuso a partir de 5.0

Variables de control internas que afectan a do/loop

Variable de control	Ámbito	Valor (valor inicial)	¿Qué controla?	Consultar /Modificar
run-sched-var	entorno de datos	(kind[,chunk)]) (depende de la implementación)	Planificación de bucles para `runtime`	sí(f) /sí(ve,f)
def-sched-var	dispositivo	(kind[,chunk)]) (depende de la implementación)	Planificación de bucles por defecto. Ámbito el programa.	no /no

2. Variables de entorno

Variable de control	Variable de entorno	Ejemplos de modificación (shell bash/ksh)
dyn-var	OMP_DYNAMIC	export OMP_DYNAMIC=FALSE export OMP_DYNAMIC=TRUE
nthreads-var	OMP_NUM _THREADS	export OMP_NUM_THREADS=8
thread-limit-var	OMP_THREAD_LIMIT	export OMP_THREAD_LIMIT=8
nest-var	OMP_NESTED	export OMP_NESTED=TRUE export OMP_NESTED=FALSE
run-sched-var	OMP_SCHEDULE	export OMP_SCHEDULE="static,4" export OMP_SCHEDULE="nonmonotonic:static,4" export OMP_SCHEDULE="dynamic" export OMP_SCHEDULE="monotonic:dynamic,4"
def-sched-var		

3. Funciones del entorno de ejecución

Variable de control	Rutina para consultar	Rutina para modificar
dyn-var	omp_get_dynamic()	omp_set_dynamic()
nthreads-var	omp_get_max_threads()	omp_set_num_threads()
thread-limit-var	omp_get_thread_limit()	
nest-var	omp_get_nested()	omp_set_nested()
run-sched-var	omp_get_schedule(&kind, &chunk)	omp_set_schedule(kind, chunk)
def-sched-var	no	no

void omp_set_schedule(omp_sched_t kind, int chunk_size);

Otras rutinas del entorno de ejecución (V2.5)

- omp_get_thread_num()
  - Devuelve al thread su identificador dentro del grupo de thread
- omp_get_num_threads()
  - Obtiene el nº de threads que se están usando en una región paralela
  - Devuelve 1 en código secuencial
- omp_get_num_procs()
  - Devuelve el nº de procesadores disponibles para el programa en el momento de la ejecución.
- omp_in_parallel()
  - Devuelve true si se llama a la rutina dentro de una región parallel activa (puede estar dentro de varios parallel, basta que uno esté activo) y false en caso contrario.

4. Cláusulas para interactuar con el entorno

¿Cuántos threads se usan?

Orden de precedencia para fijar el nº de threads, de más a menos prioridad:

- El nº resultante de evaluar la cláusula if
- El nº que fija la cláusula num_threads
- El nº que fija la función omp_set_num_threads()
- El contenido de la variable de entorno OMP_NUM_THREADS
- Fijado por defecto por la implementación: normalmente el nº de cores de un nodo

TIPO	Cláusula	Directivas					
		parallel	DO/for	sections	single	parallel DO/for	parallel sections
Control nº threads	if (1)	X				X	X
	num_threads (1)	X				X	X
Control ámbito de las variables	shared	X	X			X	X
	private	X	X	X	X	X	X
	lastprivate		X	X		X	X
	firstprivate	X	X	X	X	X	X
	default (1)	X				X	X
	reduction	X	X	X		X	X
Copia de valores	copyin	X				X	X
	copyprivate				X		
Planifica. iteraciones bucle	schedule (1)		X			X	
	ordered (1)		X			X	
No espera	nowait		X	X	X		

Cláusulas para interactuar con el entorno

- Cláusula if - no hay ejecución paralela si no se cumple la condición. Sólo en construcciones parallel.
- Cláusula schedule(kind OPT[, chunk]) - kind forma de asignación, chunk granularidad de la distribución.

Sólo bucles, kind por defecto static en la mayor parte de implementaciones. Mejor no asumir una granularidad por defecto.

- `schedule(static, chunk)` - las unidades se asignan en round robin, las iteraciones se diviten unidades de "chunk" iteraciones. Un único chunk a cada thread.
- `schedule(dynamic, chunk)` - distribución en tiempo de ejecución, apropiado si se desconoce el tiempo de ejecución de las iteraciones. La unidad de distribución tiene chunk iteraciones. Precaución, añade cierta sobrecarga.
- `schedule(guided, chunk)` - distribución en tiempo de ejecución, apropiado si se desconoce el el tiempo de ejecución de las iteraciones o su número.

  Comienza con un bloque largo. El tamaño del bloque se va reduciendo (num iteraciones restantes / num threads), no más pequeño que el chunk excepto la última. Precaución, sobrecarga extra, pero menos que dynamic.
- `schedule(runtime)` - el tipo de distribución se fija en ejecución. El tipo de depende de la variable de control run-sched-var.

5. Clasificación de las funciones de la biblioteca OpenMP

Funciones para acceder al entorno de ejecución de OpenMP:

- Funciones para usar sincronización con cerrojos:
  - V2.5 omp_init_lock(), omp_destroy_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock()
  - V3.0: omp_destroy_nest_lock, omp_set_nest_lock, omp_unset_nest_lock, omp_test_nest_lock
- Funciones para obtener tiempos de ejecución:
  - omp_get_wtime (), omp_get_wtick()

6. Funciones para obtener el tiempo de ejecución

Compilar con -lrt para incluir librería real time (no siempre es necesario)

C Pi

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

main(int argc, char **argv)
{
 register double width,x;
 double sum=0;
 register int intervals, i;

 struct timespec cgt1,cgt2;
 double ncgt;

 if(argc < 2)    {
       fprintf(stderr,"\nFalta nº intervalos\n");
       exit(-1);
 }
 intervals=atoi(argv[1]);
 if (intervals<1) intervals=1;
```

```c
 clock_gettime(CLOCK_REALTIME,&cgt1);

 width = 1.0 / intervals;

 for (i=0; i<intervals; i++) {
     x = (i + 0.5) * width;
     sum += 4.0 / (1.0 + x * x);
 }

 sum *= width;

 clock_gettime(CLOCK_REALTIME,&cgt2);
 ncgt=(double) (cgt2.tv_sec-cgt1.tv_sec)+
 (double) ((cgt2.tv_nsec-cgt1.tv_nsec)/(1.e+9));

 printf("Iteraciones:\t%d\t. PI:\t%26.24f\t.
          Threads:\t1\t. Tiempo:\t%8.6f\n",
           intervals,sum,ncgt);

 return(0);
}
```

Seminario 4 - Optimización de código en arquitecturas ILP

1. Cuestiones generales sobre la optimización

Usualmente la optimización de una aplicación se realiza al final del proceso, si queda tiempo. Esperar al final para optimizar dificulta el proceso de optimización. Es un error programar la aplicación sin tener en cuenta la arquitectura o arquitecturas en las que se va a ejecutar.

En el caso de que no se satisfagan las restricciones de tiempo, no es correcto optimizar eliminando propiedades y funciones (features) del código.

Cuando se optimiza código es importante analizar donde se encuentran los cuellos de botella. El cuello de botella más estrecho es el que al final determina las prestaciones y es el que debe evitarse en primer lugar. Se puede optimizar sin tener que acceder al nivel del lenguaje ensamblador (aunque hay situaciones en las que es necesario bajar a nivel de ensamblador).

Optimizaciones desde el Lenguaje de Alto Nivel (OHLL)		Optimizaciones desde el Lenguaje Ensamblador (OASM)	
Optimizaciones aplicables a cualquier procesador (OGP)	Optimizaciones específicas para un procesador (OEP)	Optimizaciones aplicables a cualquier procesador (OGP)	Optimizaciones específicas para un procesador (OEP)

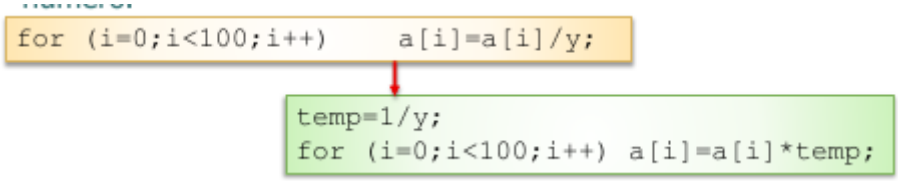
Un compilador puede ejecutarse utilizando diversas opciones de optimización. Por ejemplo, gcc/g++ dispone de las opciones -O1, - O2, -O3, -Os que proporcionan códigos con distintas opciones de optimización.

2. Optimización de la ejecución

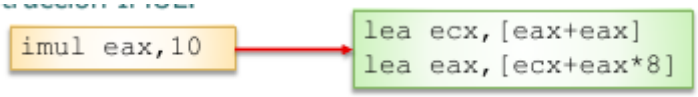
Unidades de ejecución o funcionales

Es importante tener en cuenta las unidades funcionales de que dispone la microarquitectura para utilizar las instrucciones de la forma más eficaz.

- La división es una operación costosa y por lo tanto habría que evitarla (con desplazamientos, multiplicaciones, ...) o reducir su número.



- A veces es más rápido utilizar desplazamientos y sumas para realizar una multiplicación por una constante entera que utilizar la instrucción IMUL.

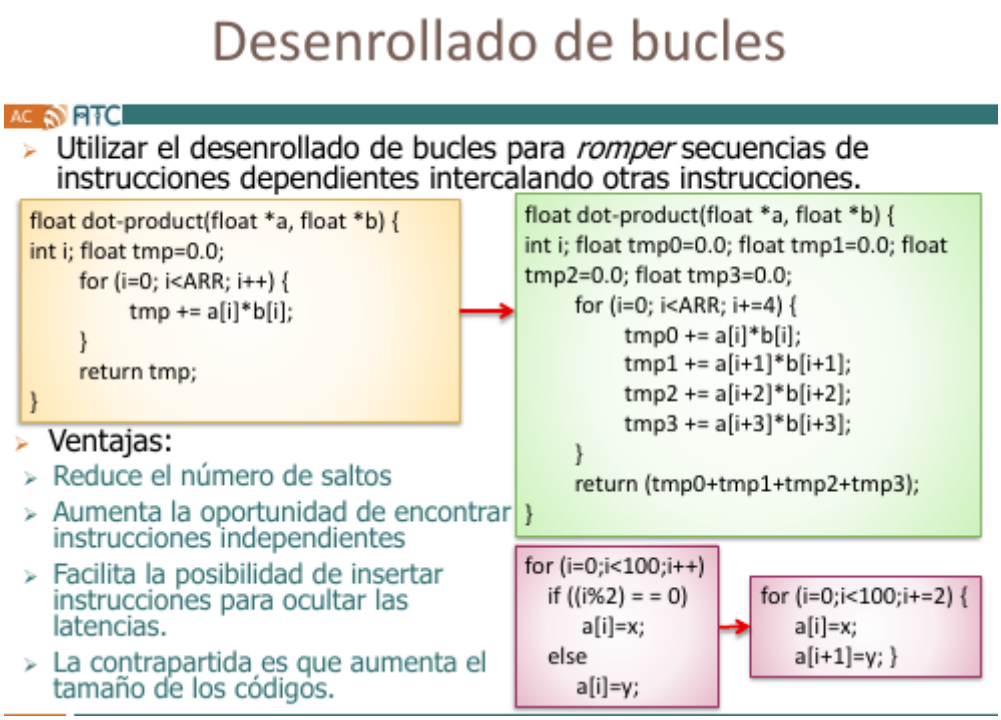


Desenrrollado de bucles

Utilizar el desenrollado de bucles para romper secuencias de instrucciones dependientes intercalando otras instrucciones. Ventajas:

- Reduce el número de saltos
- Aumenta la oportunidad de encontrar instrucciones independientes
- Facilita la posibilidad de insertar instrucciones para ocultar las latencias.

La contrapartida es que aumenta el tamaño del código.



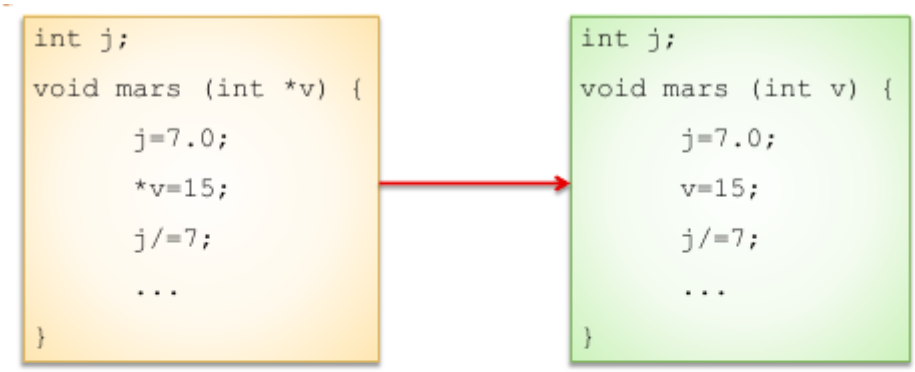
Código ambiguo

Si el compilador no puede resolver los punteros (código ambiguo) se inhiben ciertas optimizaciones del compilador:

- Asignar variables durante la compilación
- Realizar cargas de memoria mientras que un almacenamiento está en marcha

Si no se utilizan punteros el código es más dependiente de la máquina, y a veces las ventajas de no utilizarlos no compensa. ¿Cómo evitar código ambiguo o sus efectos?:

- Utilizar variables locales en lugar de punteros
- Utilizar variables globales si no se pueden utilizar las locales
- Poner las instrucciones de almacenamiento después o bastante antes de las de carga de memoria.

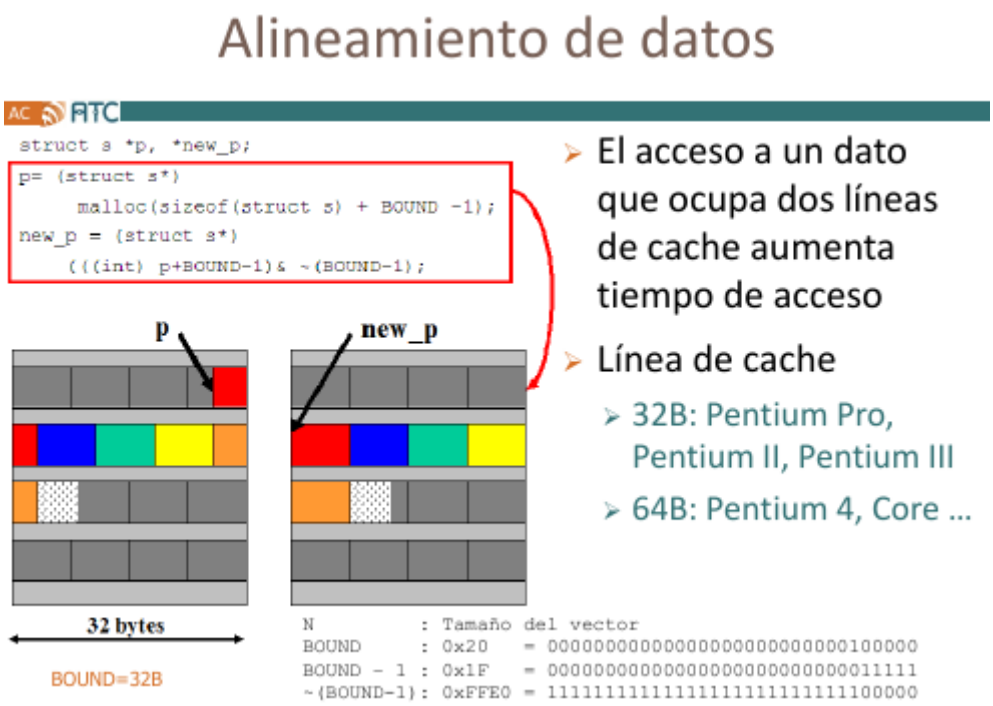


- En el código no optimizado, el compilador no puede asumir que *v no apunta a j.
- Sin embargo en el código optimizado, el compilador puede hacer directamente j=1.0.

4. Optimización del acceso a memoria

Alineamiento de datos

El acceso a un dato que ocupa dos líneas de cache aumenta tiempo de acceso. Tam línea de cache: 32B en Pentium Pro, Pentium II, Pentium III; 64B en Pentium 4, Core ...



Colisiones en caché

Típicamente las caches tienen una organización asociativa por conjuntos:

- L1 Intel Core 2 Duo, Intel Core Duo, Intel Core Solo:
  - Líneas cache 64B, asociativa por conjuntos 8 vías, tamaño 32 KB
  - Penalización si se accede a más de 8 direcciones con separación de 4KB (= 32KB/8 vías)

```
int *tempA, *tempB;
...
pA= (int *) malloc (sizeof(int)*N + 63);
tempA = (int *)(((int)pA+63)&~(63));
tempB = (int *)((((int)pA+63)&~(63))+4096+64);
```

Los punteros tempA y tempB están apuntando a zonas de memoria que empiezan en posiciones que son múltiplos de 64 y que no se asignaránn al mismo conjunto de cache L1

- L2 Intel Core 2 Duo, Intel Core Duo, Intel Core Solo:
  - Penalización si se accede a más de X direcciones con separación de 256KB (X: nº de vías de la cache L2)

Localidad de los accesos

La forma en que se declaren los arrays determina la forma en que se almacenan en memoria. Interesa declararlos según la forma en la que se vaya a realizar el acceso. Ejemplos: Formas óptimas de declaración de variables según el tipo de acceso a los datos:

```
struct {
      int a[500];
      int b[500];
           } s;
...
for (i=0; i<500; i++)
      s.a[i]=2*s.a[i];
...
for (i=0;i<500;i++)
      s.b[i]=3*s.b[i];
```

```
struct {
      int a;
      int b;
           } s[500];
…
for (i=0;i<500;i++)
{
      s[i].a+=5;
      s[i].b+=3;
}
```

Intercambiar los bucles para cambiar la forma de acceder a los datos según los almacena el compilador, y para aprovechar la localidad. Ej:

Código Original

```
for (j=0; j<4000; j++)
      for (i=0;i<4000;i++)
            a[i][j]=2*a[i][j];
```

Código Optimizado para C (se almacena la matriz por filas)

```
for (i=0; i<4000; i++)
      for (j=0;j<4000;j++)
            a[i][j]=2*a[i][j];
```

Acceso a memoria especulativo

Los 'atascos' (stalls) por acceso a la memoria (load adelanta a store, especulativo) se producen cuando:

1. Hay una carga (load) 'larga' que sigue a un almacenamiento (store) 'pequeño' alineados en la misma dirección o en rangos de direcciones solapadas.

```
mov word ptr [ebp],0x10
```

```
mov ecx, dword ptr [ebp]
```

2. Una carga (load) 'pequeña' sigue a un almacenamiento (store) 'largo' en direcciones diferentes aunque solapadas (si están alineadas en la misma dirección no hay problema).

```
mov dword ptr [ebp-1], eax
```

```
mov ecx,word ptr [ebp]
```

3. Datos del mismo tamaño se almacenan y luego se cargan desde direcciones solapadas que no están alineadas.

```
mov dword ptr [ebp-1], eax
```

```
mov eax, dword ptr [ebp]
```

Para evitarlos: utilizar datos del mismo tamaño y direcciones alineadas y poner los loads tan lejos como sea posible de los stores a la misma área de memoria

Precaptación (Prefetch)

El procesador, mediante las correspondientes instrucciones de prefetch, carga zonas de memoria en cache antes de que se soliciten (cuando hay ancho de banda disponible). Hay cuatro tipos de instrucciones de prefetch:

Instrucción ensamb.	Segundo parámetro en `_mm_prefetch()` (Intel C++ Compiler)	Descripción
prefetchnta	_MM_HINT_NTA	Prefetch en buffer no temporal (dato para una lectura)
prefetcht0	_MM_HINT_T0	Prefetch en todas las caches útiles
prefetcht1	_MM_HINT_T1	Prefetch en L2 y L3 pero no en L1
prefetcht2	_MM_HINT_T2	Prefetch sólo en L3

- En gcc se puede usar:
  - void __builtin_prefetch (const void *addr, ...)

Una instrucción de prefetch carga una línea entera de cache. El aspecto crucial al realizar precaptación es la anticipación con la que se pre-captan los datos. En muchos casos, es necesario aplicar una estrategia de prueba y error. Además, la anticipación óptima puede cambiar según las características del computador (menos portabilidad en el código).

Ejemplo:

```
for (i=0; i<1000; i++) {
        x=function(matriz[i]);
        _mm_prefetch(matriz[i+16],_MM_HINT_T0);
}
```

- En el ejemplo se precapta el dato necesario para la iteración situada a 16 iteraciones (en el futuro)
- En un prefetch no se generan faltas de memoria (es seguro precaptar más allá de los límites del *array*)

5. Optimización de saltos

Saltos I

```
if  (t1==0 && t2==0 && t3==0)
```

- Cada una de las condiciones separadas por && se evalúa mediante una instrucción de salto distinta.
  - Si las variables pueden ser 1 ó 0 con la misma probabilidad, la posibilidad de predecir esas instrucciones de salto no es muy elevada.

```
if  ((t1 | t2 | t3)==0)
```

- Si se utiliza un único salto, la probabilidad de 1 es de 0.125 y la de 0 de 0.875 y la posibilidad de hacer una buena predicción aumenta.
- Mediante la instrucción de movimiento condicional se pueden evitar los saltos

```
//if ((t1 | t2 | t3)==0) {t4=1};
mov    ecx, 1    //t1 -> edi
or     edi, ebx  //t2 -> ebx
or     edi, ebp  //t3 -> ebp
cmove  eax, ecx  //t4 -> eax
```

Saltos II

- Se puede reducir el número de saltos de un programa reorganizando las alternativas en las sentencias *switch*, en el caso de que alguna opción se ejecute mucho más que las otras (más del 50% de las veces, por ejemplo).
- Ciertos compiladores que utilizan información de perfiles de ejecución del programa son capaces de realizar esta reorganización (Se recomienda utilizarla si la sentencia *switch* se implementa como una búsqueda binaria en lugar de una tabla de salto).

Código original

```
switch (i) {
 case 16:
      Bloque16
      break;
 case 22:
      Bloque22
      break;
 case 33:
      Bloque33
      break;
}
```

```
if (i==33)
      { Bloque33 }
else
switch (i) {
      case 16:
           Bloque16
           break;
      case 22:
           Bloque22
           break;
}
```

Código Optimizado

Saltos III

- CMOVcc hace la transferencia de información si se cumple la condición indicada en cc

```
test ecx,ecx
jne 1h
mov eax,ebx
1h:
```

```
test ecx,ecx
cmoveq eax, ebx
```

- FCMOVcc es similar a CMOVcc pero utiliza operandos en coma flotante

Saltos IV

AC ATC

- La instrucción `SETcc` es otro ejemplo de instrucción con predicado que puede permitir reducir el número de instrucciones de salto.

Código Original

```
        cmp  A,B
        jge   L30
        mov  ebx,C1
        jmp   L31
L30:    mov  ebx,C2
L31:
```

ebx= (A<B) ? C1 : C2; [Si (A<B) es cierto EBX se carga con C1 y si no con C2]

Código Optimizado

```
xor ebx,ebx
cmp A,B
setge bl
dec ebx
and ebx, (C1-C2)
add ebx, C2
```

Explicación:

```
//Si A>=B, setge hace BL=1; DEC hace EBX=0; (EBX and C1-C2)=0;
//EBX + C2 = C2
//Si A<B, setge deja BL=0; DEC hace EBX=0xFFFFFFFF;
//(EBX and C1-C2)= C1-C2;
//EBX + C2 = C1
```