

Plataformas de soporte computacional: Sistemas de memoria compartida

Diego R. Llanos, Belén Palop
Departamento de Informática
Universidad de Valladolid
{diego,b.palop}@infor.uva.es



Universidad de Valladolid

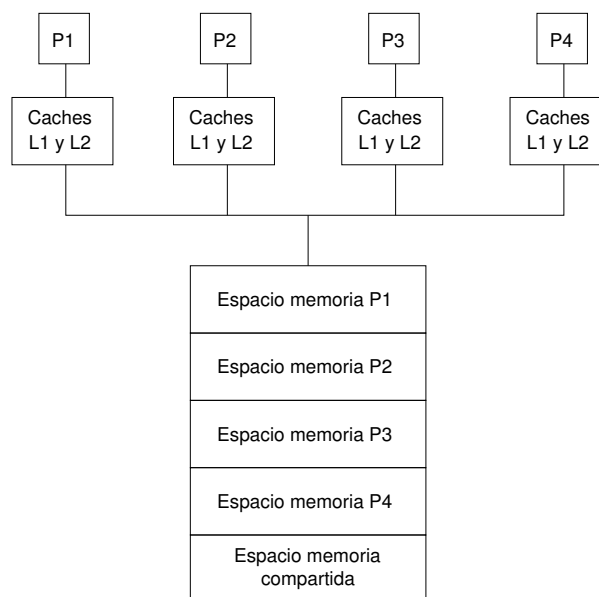
Índice

1. Sistemas paralelos	1
2. El estándar OpenMP	3
3. OpenMP reference card	6

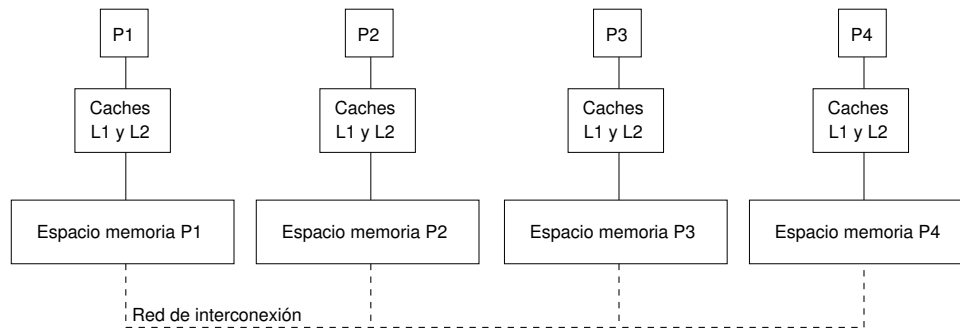
1. Sistemas paralelos

Arquitectura de sistemas paralelos

- *Sistemas de memoria compartida:* Todos los procesadores comparten una zona de memoria. Los datos escritos en ella por un procesador son visibles al resto. Utilizan un modelo de programación “de variables compartidas”.



- *Sistemas de memoria distribuida*: No existe un espacio de memoria compartida: la comunicación se realiza mediante “paso de mensajes”.

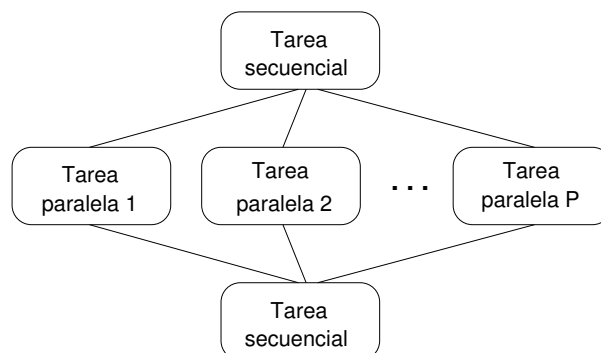


Arquitectura de sistemas de memoria compartida

- Premisa básica: varios procesadores, un único espacio de memoria direccionable.
- En una primera aproximación, puede ignorarse la jerarquía de memoria intermedia (cachés).
- En este tema veremos el modelo de programación OpenMP, basado en el uso de variables compartidas.

Modelo de programación de variables compartidas

- Caracterizado porque todos los procesadores comparten un espacio común de direccionamiento: las variables declaradas como *compartidas* (“shared”). Existen otras variables propias de cada procesador: variables *privadas* (“private”).
- Este modelo de programación se basa en el llamado *paralelismo anidado*: un procesador lanza una región paralela, y cuando todos los procesadores de la región terminan, se continúa con la ejecución secuencial.



Una advertencia: la Ley de Amhdahl

- La Ley de Amhdahl limita la eficacia de la ejecución paralela de una aplicación.
- Si T_S es la fracción del tiempo en el que se realizan tareas secuenciales en un programa, el speedup máximo obtenible con P procesadores es el siguiente:

$$\text{speedup} = \frac{1}{T_S + \frac{1-T_S}{P}}$$

- Así, si T_S fuera 0,5, contaríamos con *infinitos* procesadores y no hubiera costes de sincronización, ¡el speedup máximo sería de dos!

Implementación de un modelo de memoria compartida

- Desde el punto de vista de la programación, hacen falta tres tipos de servicios del sistema operativo:
 - Creación/destrucción de procesos (primitivas UNIX de tipo `fork()`). A partir de un cierto punto, el proceso actual se replica en varios procesos, distinguibles sólo por su identificador de proceso (PID).
 - Definición de un espacio de memoria compartida (primitivas UNIX `shmget()`, `shmat()` y `shmdt()`). El manejo se hace a través de punteros.
 - Uso de primitivas de sincronización (como semáforos, secciones críticas o barreras) para regular el acceso a la memoria compartida y señalar la finalización de un proceso paralelo.
- Toda la gestión de los procesos y su sincronización corren por cuenta del usuario: la programación paralela siguiendo este método no es muy popular.

2. El estándar OpenMP

El estándar OpenMP

- Se trata de una capa de abstracción que oculta muchos de los detalles “desagradables” de la programación paralela en sistemas de memoria compartida.
- La idea básica es desarrollar un código secuencial que resuelva la tarea, y luego aplicar las directivas OpenMP para paralelizarlo.
- La corrección de las tareas que se están realizando corren a cuenta del programador: OpenMP sólo oculta los detalles de la implementación.
- Ejemplo: Sea el siguiente programa secuencial en lenguaje C:

```
main(){
    int vector[10],i;
    for (i=0;i<10;i++)
        vector[i]=i;
    printf("Hecho.\n");
}
```

- Es fácil ver que el valor que recibe `vector[i]` es independiente de los valores de sus vecinos. Informalmente, se diría que este código no tiene *dependencias* y que por lo tanto puede ejecutarse en paralelo.
- La versión paralela haría las diez asignaciones simultáneamente:

```
#include<omp.h>
main(){
    int vector[10],i;
    omp_set_num_threads(10);
    #pragma omp parallel for \
        private (i) shared (vector) \
        schedule (static)
    for (i=0;i<10;i++)
        vector[i]=i;
    printf("Hecho.\n");
}
```

La directiva `omp parallel for`

- Esta directiva permite ejecutar un bucle `for` en paralelo.
- Algunos argumentos:
 - **private**: Permite enumerar las variables que se considerarán *privadas*, es decir que estarán almacenadas en el espacio de memoria propio de cada proceso.
 - **shared**: Permite enumerar las variables que se considerarán *compartidas* por todos los procesos.
 - **schedule**: permite indicar cómo se repartirá el trabajo. El parámetro **static** indica que cada thread recibirá una parte proporcional (aquí, una iteración).
 - **default**: Permite indicar el comportamiento por defecto de las variables no listadas como **shared** ni **private**. *Hay que poner siempre `default(none)`.*

El problema de las dependencias

- El problema es que muchas iteraciones dependen del resultado generado en la iteración anterior:

```
main(){
    int vector[10],i,a=0;
    for (i=0;i<10;i++) {
        vector[i]=i;
        a=a+vector[i];
    }
    printf("Hecho.\n");
}
```

- En este ejemplo, la variable *a* es compartida, pero su valor depende del resultado de la iteración anterior.
- Algunas dependencias se pueden tratar: la expresión `variable=variable+algo` se denomina “reducción de suma”.
- OpenMP es capaz de acumular todos los resultados parciales y sumarlos al final, cuando el bucle termina, gracias a que la suma es asociativa. Lo mismo pasa con la multiplicación y con otros operadores.

```
#include<omp.h>
main(){
    int vector[10],i,a=0;
    omp_set_num_threads(10);
    #pragma omp parallel for \
        private (i) shared (vector,a) reduction (+:a) \
        schedule (static)
    for (i=0;i<10;i++) {
        vector[i]=i;
        a=a+vector[i];
    }
    printf("Hecho.\n");
}
```

La directiva `omp critical`

- Esta directiva permite definir secciones críticas: fragmentos de código en donde sólo puede estar trabajando un proceso a la vez:
- Supongamos que tenemos el siguiente programa:

```

#include<omp.h>
main(){
    int i,a=0;
    #pragma omp parallel for \
        private (i) shared (a) schedule (static)
    for (i=0;i<2;i++) {
        a=a+1;
    }
}

```

Si dos procesos cooperaran en esta tarea, la secuencia de eventos podría ser la siguiente:

1. El proceso 1 lee el valor de la variable **a** (que vale cero).
 2. El proceso 2 lee el valor de la variable **a** (que vale cero).
 3. El proceso 1 calcula **a+1**, obteniendo un uno.
 4. El proceso 1 escribe el resultado en la variable **a**.
 5. El proceso 2 calcula **a+1**, obteniendo un uno.
 6. El proceso 2 escribe el resultado en la variable **a**.
- La directiva `omp critical` evita esta clase de “solapamientos” en la ejecución de un fragmento de código:

```

#include<omp.h>
main(){
    int i,a=0;
    #pragma omp parallel for \
        private (i) shared (a) schedule (static)
    for (i=0;i<2;i++) {
        #pragma omp critical
        a=a+1;
        #pragma omp end critical
    }
}

```

- Si sólo se trata de proteger una instrucción, existe también la directiva `omp atomic`, que convierte la ejecución de la instrucción en indivisible.

Ventajas de OpenMP


- Todo el trabajo de definir **vector** como una variable compartida, crear los threads, lanzarlos, y esperar a que todos terminen para continuar queda oculto al programador.
- Esto ha convertido a OpenMP en un estándar de facto en el campo de la programación paralela en ingeniería.
- Hoy día, todos los compiladores de dominio público y propietarios incluyen soporte para la compilación con directivas OpenMP. Por ejemplo, en GCC puede compilarse un programa con OpenMP como:

```
gcc programa.c -o programa -fopenmp
```

- Recordar que la eficacia de la implementación dependerá de cuántos núcleos tenga nuestro sistema paralelo!

3. OpenMP reference card

Summary of OpenMP 3.0 C/C++ Syntax



Download the full OpenMP API Specification at www.openmp.org.

Directives

An OpenMP executable directive applies to the succeeding structured block or an OpenMP Construct. A "structured block" is a single statement or a compound statement with a single entry at the top and a single exit at the bottom.

The **parallel** construct forms a team of threads and starts parallel execution.

```
#pragma omp parallel [clause[ [, ]clause] ...] new-line
structured-block
```

clause: **if** (*scalar-expression*)
num_threads (*integer-expression*)
default (**shared** | **none**)
private (*list*)
firstprivate (*list*)
shared (*list*)
copyin (*list*)
reduction (*operator* : *list*)

The loop construct specifies that the iterations of loops will be distributed among and executed by the encountering team of threads.

```
#pragma omp for [clause[[ , ]clause] ...] new-line
for-loops
```

clause: **private** (*list*)
firstprivate (*list*)
lastprivate (*list*)
reduction (*operator* : *list*)
schedule (*kind*, *chunk_size*)
collapse (*n*)
ordered
nowait

The most common form of the for loop is shown below.

```
for(var = lb;
var relational-op b;
var += incr)
```

The **sections** construct contains a set of structured blocks that are to be distributed among and executed by the encountering team of threads.

```
#pragma omp sections [clause[[ , ]clause] ...] new-line
{
  [#pragma omp section new-line]
  structured-block
  [#pragma omp section new-line]
  structured-block
  ...
} (See applicable clauses on next page.)
```

1

Directives (continued)

clause: **private** (*list*)
firstprivate (*list*)
lastprivate (*list*)
reduction (*operator* : *list*)
nowait

The **single** construct specifies that the associated structured block is executed by only one of the threads in the team (not necessarily the master thread), in the context of its implicit task.

```
#pragma omp single [clause[[ , ]clause] ...] new-line
structured-block
```

clause: **private** (*list*)
firstprivate (*list*)
copyprivate (*list*)
nowait

The combined parallel worksharing constructs are a shortcut for specifying a parallel construct containing one worksharing construct and no other statements. Permitted clauses are the union of the clauses allowed for the **parallel** and worksharing constructs.

```
#pragma omp parallel for [clause[[ , ]clause] ...] new-line
for-loop
```

```
#pragma omp parallel sections [clause[[ , ]clause] ...]
new-line
{
  [#pragma omp section new-line]
  structured-block
  [#pragma omp section new-line]
  structured-block
  ...
}
```

The **task** construct defines an explicit task. The data environment of the task is created according to the data-sharing attribute clauses on the task construct and any defaults that apply.

```
#pragma omp task [clause[ [, ]clause] ...] new-line
structured-block
```

clause: **if** (*scalar-expression*)
untied
default (**shared** | **none**)
private (*list*)
firstprivate (*list*)
shared (*list*)

The **master** construct specifies a structured block that is executed by the master thread of the team. There is no implied barrier either on entry to, or exit from, the master construct.

```
#pragma omp master new-line
structured-block
```

2

Directives (continued)

The **critical** construct restricts execution of the associated structured block to a single thread at a time.

```
#pragma omp critical [(name)] new-line
structured-block
```

The **barrier** construct specifies an explicit barrier at the point at which the construct appears.

```
#pragma omp barrier new-line
```

The **taskwait** construct specifies a wait on the completion of child tasks generated since the beginning of the current task.

```
#pragma omp taskwait new-line
```

The **atomic** construct ensures that a specific storage location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.

```
#pragma omp atomic new-line
expression-stmt
```

expression-stmt: one of the following forms:
x binop = expr
x++
++x
x--
--x

The **flush** construct executes the OpenMP flush operation, which makes a thread's temporary view of memory consistent with memory, and enforces an order on the memory operations of the variables.

```
#pragma omp flush [(list)] new-line
```

The **ordered** construct specifies a structured block in a loop region that will be executed in the order of the loop iterations. This sequentializes and orders the code within an ordered region while allowing code outside the region to run in parallel.

```
#pragma omp ordered new-line
structured-block
```

The **threadprivate** directive specifies that variables are replicated, with each thread having its own copy.

```
#pragma omp threadprivate(list) new-line
```

3

Clauses

Not all of the clauses are valid on all directives. The set of clauses that is valid on a particular directive is described with the directive. Most of the clauses accept a comma-separated list of list items. All list items appearing in a clause must be visible.

Data Sharing Attribute Clauses

Data-sharing attribute clauses apply only to variables whose names are visible in the construct on which the clause appears.

default (**shared** | **none**) ;
 Controls the default data-sharing attributes of variables that are referenced in a **parallel** or **task** construct.

shared (*list*) ;
 Declares one or more list items to be shared by tasks generated by a **parallel** or **task** construct.

private (*list*) ;
 Declares one or more list items to be private to a task.

firstprivate (*list*) ;
 Declares one or more list items to be private to a task, and initializes each of them with the value that the corresponding original item has when the construct is encountered.

lastprivate (*list*) ;
 Declares one or more list items to be private to an implicit task, and causes the corresponding original item to be updated after the end of the region.

reduction (*operator* : *list*) ;
 Declares accumulation into the list items using the indicated associative operator. Accumulation occurs into a private copy for each list item which is then combined with the original item.

Data Copying Clauses

These clauses support the copying of data values from private or thread-private variables on one implicit task or thread to the corresponding variables on other implicit tasks or threads in the team.

copyin (*list*) ;
 Copies the value of the master thread's *threadprivate* variable to the *threadprivate* variable of each other member of the team executing the **parallel** region.

copyprivate (*list*) ;
 Broadcasts a value from the data environment of one implicit task to the data environments of the other implicit tasks belonging to the **parallel** region.

4

Runtime Library Routines

Execution environment routines affect and monitor threads, processors, and the parallel environment. Lock routines support synchronization with OpenMP locks. Timing routines support a portable wall clock timer. Prototypes for the runtime library routines are defined in the file "omp.h".

Execution Environment Routines

```
void omp_set_num_threads(int num_threads);
Affects the number of threads used for subsequent parallel
regions that do not specify a num_threads clause.

int omp_get_num_threads(void);
Returns the number of threads in the current team.

int omp_get_max_threads(void);
Returns maximum number of threads that could be used to form a new
team using a "parallel" construct without a "num_threads" clause.

int omp_get_thread_num(void);
Returns the ID of the encountering thread where ID ranges from zero
to the size of the team minus 1.

int omp_get_num_procs(void);
Returns the number of processors available to the program.

int omp_in_parallel(void);
Returns true if the call to the routine is enclosed by an active
parallel region; otherwise, it returns false.

void omp_set_dynamic(int dynamic_threads);
Enables or disables dynamic adjustment of the number of threads
available.

int omp_get_dynamic(void);
Returns the value of the dyn-var internal control variable (ICV),
determining whether dynamic adjustment of the number of threads is
enabled or disabled.

void omp_set_nested(int nested);
Enables or disables nested parallelism, by setting the nest-var ICV.

int omp_get_nested(void);
Returns the value of the nest-var ICV, which determines if nested
parallelism is enabled or disabled.

void omp_set_schedule(omp_sched_t kind, int modifier);
Affects the schedule that is applied when runtime is used as
schedule kind, by setting the value of the run-sched-var ICV.

void omp_get_schedule(omp_sched_t *kind,
int *modifier);
Returns the schedule applied when runtime schedule is used.
```

5

Runtime Library Routines (continued)

```
int omp_get_thread_limit(void)
Returns the maximum number of OpenMP threads available to the
program.

void omp_set_max_active_levels(int max_levels);
Limits the number of nested active parallel regions, by setting the
max-active-levels-var ICV.

int omp_get_max_active_levels(void);
Returns the value of the max-active-levels-var ICV, which determines
the maximum number of nested active parallel regions.

int omp_get_level(void);
Returns the number of nested parallel regions enclosing the task
that contains the call.

int omp_get_ancestor_thread_num(int level);
Returns, for a given nested level of the current thread, the thread
number of the ancestor or the current thread.

int omp_get_team_size(int level);
Returns, for a given nested level of the current thread, the size of the
thread team to which the ancestor or the current thread belongs.

int omp_get_active_level(void);
Returns the number of nested, active parallel regions
enclosing the task that contains the call.
```

Lock Routines

```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
These routines initialize an OpenMP lock.

void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
These routines ensure that the OpenMP lock is uninitialized.

void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
These routines provide a means of setting an OpenMP lock.

void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
These routines provide a means of setting an OpenMP lock.

int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_nest_lock_t *lock);
These routines attempt to set an OpenMP lock but do not suspend
execution of the task executing the routine.
```

6

Runtime Library Routines (continued)

Timing Routines

```
double omp_get_wtime(void);
Returns elapsed wall clock time in seconds.

double omp_get_wtick(void);
Returns the precision of the timer used by omp_get_wtime.
```

Environment Variables

Environment variable names are upper case, and the values assigned to them are case insensitive and may have leading and trailing white space.

```
OMP_SCHEDULE type[, chunk]
Sets the run-sched-var ICV for the runtime schedule type and chunk
size. Valid OpenMP schedule types are static, dynamic, guided, or
auto. Chunk is a positive integer.

OMP_NUM_THREADS num
Sets the nthreads-var ICV for the number of threads to use for
parallel regions.

OMP_DYNAMIC dynamic
Sets the dyn-var ICV for the dynamic adjustment of threads to use for
parallel regions. Valid values for dynamic are true or false.

OMP_NESTED nested
Sets the nest-var ICV to enable or to disable nested parallelism. Valid
values for nested are true or false.

OMP_STACKSIZE size
Sets the stacksize-var ICV that specifies the size of the stack for
threads created by the OpenMP implementation. Valid values for size
(a positive integer) are size, sizeB, sizeK, sizeM, sizeG. If units B, K, M
or G are not specified, size is measured in kilobytes (K).

OMP_WAIT_POLICY policy
Sets the wait-policy-var ICV that controls the desired behavior of
waiting threads. Valid values for policy are active (waiting threads
consume processor cycles while waiting) and passive.

OMP_MAX_ACTIVE_LEVELS levels
Sets the max-active-levels-var ICV that controls the maximum
number of nested active parallel regions.

OMP_THREAD_LIMIT limit
Sets the thread-limit-var ICV that controls the maximum number of
threads participating in the OpenMP program.
```

7

Details

Operators legally allowed in a reduction

Operator	Initialization value
+	0
*	1
-	0
&	~0
	0
^	0
&&	1
	0

Schedule types for the loop construct

```
static Iterations are divided into chunks of size chunk_size, and
the chunks are assigned to the threads in the team in a
round-robin fashion in the order of the thread number.

dynamic Each thread executes a chunk of iterations, then requests
another chunk, until no chunks remain to be distributed.

guided Each thread executes a chunk of iterations, then requests
another chunk, until no chunks remain to be assigned.
The chunk sizes start large and shrink to the indicated
chunk_size as chunks are scheduled.

auto The decision regarding scheduling is delegated to the
compiler and/or runtime system.

runtime The schedule and chunk size are taken from the
run-sched-var ICV.
```

Copyright © 1997-2008 OpenMP Architecture Review Board. Permission to copy without fee all or part of this material is granted, provided the OpenMP Architecture Review Board copyright notice and the title of this document appear. Notice is given that copying is by permission of the OpenMP Architecture Review Board. Products or publications based on one or more of the OpenMP specifications must acknowledge the copyright by displaying the following statement: "OpenMP is a trademark of the OpenMP Architecture Review Board. Portions of this product/publication may have been derived from the OpenMP Language Application Program Interface Specification."

Rev 1108-001

8