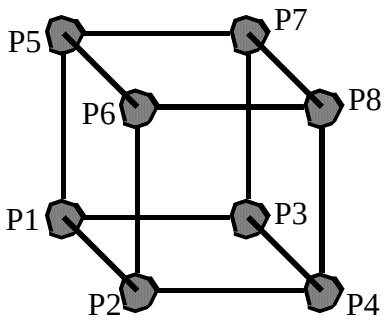
UNIVERSIDAD CARLOS III DE MADRID
DEPARTAMENTO DE INFORMÁTICA
INGENIERÍA EN INFORMÁTICA. ARQUITECTURA DE COMPUTADORES II
10 junio de 2006

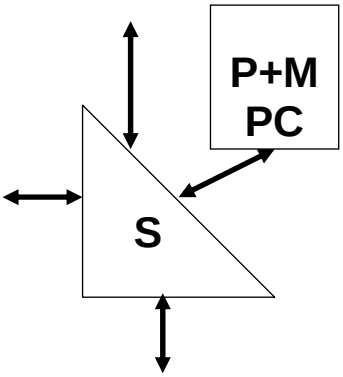
Para la realización del presente examen se dispondrá de **2 horas 30 minutos. NO** se podrán utilizar libros ni apuntes.
Entregar cada ejercicio en hojas separadas.

Problema 1. 3 Puntos.

Sea un multiprocesador de memoria distribuida con las siguientes características:

- 8 procesadores con una potencia de 10 MFLOPS (millones de operaciones en punto flotante por segundo).
- Una red de topología 3D-CUBE con 8 nodos (mostrados en la figura). Cada nodo cuenta con un procesador (P), una memoria local (M), un procesador de comunicaciones (PC) y un *switch* de comunicaciones (S).





- La red tiene las siguientes características:
  - No existe restricción en el tamaño de paquete (cualquier conjunto de datos puede ser enviado en un único paquete).
  - Protocolo de encaminamiento del *switch*: *store and forward*.
  - Ancho de banda: 160 10^6 bits/seg.
  - *Routing delay* (retardo de encaminamiento del *switch*): 0.2 segs.
  - Retardo de envío y recepción del procesador de comunicaciones: 0 ms.

Se desea paralelizar el siguiente lazo, donde **Y** es un vector de números en punto flotante de 16 bits y n=80.000.000.

```
for ( i = 0; i < n; i++)
{
     Y[i] = Y[i] + 5;
}
```

Inicialmente Y está en el procesador P1, la estrategia consiste en distribuir Y en bloques de igual tamaño entre todos los procesadores y ejecutar el lazo en paralelo.
Se pide:

1. Asumiendo una distribución del vector Y en la que el procesador P1 manda sucesivamente a cada procesador **la porción de Y que tiene asignada mediante un envío síncrono,** es decir, no se envía un nuevo mensaje hasta que el nodo destino no recibe el dato. Calcular el tiempo de distribución de los datos.
2. Definir el diámetro de la red e indicar su valor.
3. Para la distribución por bloques del apartado anterior, calcular la máxima aceleración (*speedup*) alcanzable suponiendo que existe una **barrera de sincronización** entre la fase de distribución y la de cómputo. Asumir que el control del lazo no tiene coste.
4. Una posibilidad de mejorar la aceleración consiste en realizar comunicaciones asíncronas de modo que P1 pueda enviar de modo simultáneo distintos paquetes a través de distintos caminos de la red. Calcular la orquestación más adecuada para maximizar la aceleración **asumiendo que cada *switch* sólo puede estar ocupado recibiendo un único paquete.**

Problema 2. 2 Puntos.

Dado el siguiente código de OpenMP:

```
#pragma omp parallel shared(A,RESULT,n) private(TMP)
{
     #pragma omp for
     for ( i = 0; i < n; i++)
     {
          TMP = TMP + A[i];
     }

     #pragma omp critical
     for ( j = 0; j < n/10; j++)
     {
          RESULT[j] = RESULT[j] + TMP;
     }
}
```

El tiempo de ejecución de cada iteración de cualquiera de los dos lazos es de 1ms.

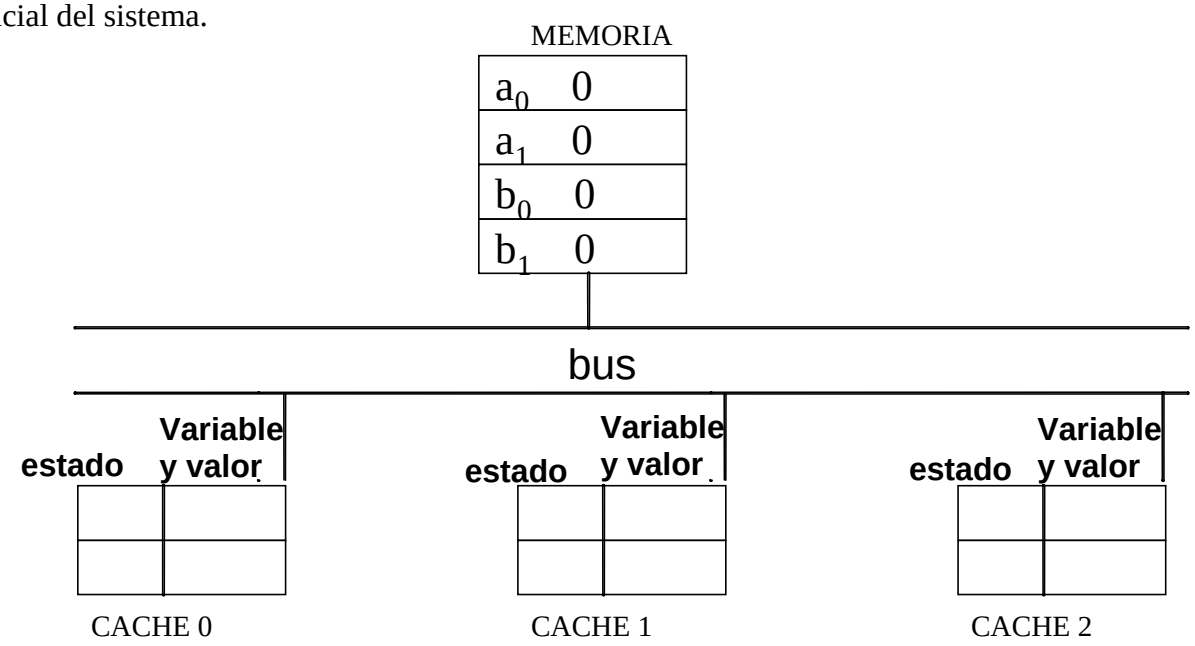
Se pide:

1. Se dispone de tres versiones distintas del código con valores de n diferentes (n=100, n=400 y n=800). Para cada una de ellas de quiere calcular la aceleración (*speedup*) teórica para los siguientes números de threads:

   Para la versión de n=200 con nthreads=2
   Para la versión de n=400 con nthreads=4
   Para la versión de n=800 con nthreads=8
2. Definir la eficiencia. Calcular la eficiencia para cada uno de los valores anteriores.
3. ¿Es un código escalable?

Problema 3. 2 Puntos.

Un computador paralelo consta de una memoria compartida que tiene 4 bloques, cada bloque consta de 4 bytes y en cada bloque hay una sola variable de tipo entero inicializada a 0. La memoria está conectada a 3 procesadores a través de un bus. Cada procesador tiene asociada una memoria caché de 2 líneas, cada línea es de 4 bytes, las cachés utilizan correspondencia directa. La siguiente figura muestra el estado inicial del sistema.



La coherencia de las memorias caché se lleva a cabo utilizando el protocolo *write-once*. A continuación se dan tres secuencias de acciones realizadas por los procesadores P_0, P_1 y P_2.

Secuencia 1

t_0: P_0 lee a_0; P_1 lee a_0 (en el tiempo 0 el proceso P_0 lee a_0 y el proceso P_1 lee a_0)
t_1: P_0 $a_0 = 5$; P_2 $a_1 = 10$

Secuencia 2

t_2: P_0 $a_0 = 15$; P_2 $a_1 = 20$

Secuencia 3

t_3: P_0 lee a_0; P_1 lee a_1

Secuencia 4

t_4: P_0 lee b_0; P_1 lee b_0

Se pide indicar el contenido de la memoria y las tres cachés (para cada línea de cada caché muestre el estado, el nombre de la variable que contiene y el valor de la misma) después de ejecutar cada una de las secuencias de instrucciones.

Problema 4. 3 Puntos.

El siguiente programa utiliza *threads* de POSIX y está incompleto. Hace uso de una barrera en la *función trabajo* y falta escribir el código de las funciones que manejan la barrera, i.e. las funciones:

- void init_barrera() que se encarga de inicializar las variables de la variable barrera_var.
- void destroy_barrera() que se encarga de destruir las variables de sincronización contenidas en la variable barrera_var.
- void barrera() que se encarga de suspender la ejecución de un proceso hasta tanto todos los demás lleguen a la barrera.

Programe las funciones:

- void init_barrera()
- void destroy_barrera()
- void barrera()

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

struct barrera_t
{
        pthread_mutex_t mutex;
        pthread_cond_t  cond;
        int contador;
};

void init_vector(int *a, int n);
void print_a(int *a, int n);
void* trabajo(void *vid);
void barrera();
void init_barrera();
void destroy_barrera();

int nprocs, tamanio_a;
int *a;
struct barrera_t barrera_var;

int main(int argc, char *argv[])
{
        pthread_t *l_threads;
        int *v;
        int i;

        if (argc != 3){
                printf("Uso: prueba Numero_de_Hilos Tamanio_a\n");
                exit(1);
        }
        nprocs = atoi(argv[1]);
        tamanio_a = atoi(argv[2]);
        if ((tamanio_a % nprocs) != 0)
                printf("Advertencia: parte de a no sera tratada\n");
        l_threads = (pthread_t *) malloc(sizeof(pthread_t) * nprocs);
        a = (int *) malloc(sizeof(int) * (tamanio_a + 1));
        v = (int *) malloc(sizeof(int) * nprocs);
        init_vector(a, tamanio_a);
        init_vector(v, nprocs);
        init_barrera();
```

```
	for (i = 0; i < nprocs; i++)
		pthread_create(&l_threads[i], NULL, &trabajo, (void *)i);

	for (i = 0; i < nprocs; i++)
		pthread_join(l_threads[i], NULL);
	destroy_barrera();

	print_a(a, tamanio_a);
	exit(0);
}

void init_vector(int *a, int n)
{
 int i;
 for (i = 0; i < n; i++)
	a[i] = i;
 a[n] = 0;
}

void print_a(int *a, int n)
{
	int i;
	for(i = 0; i < n; i++)
		printf("a[%d] = %d\n", i, a[i]);
}

void* trabajo(void * vid)
{
	int id;
	int min, max;
	int s;
	int i;

	id = (int) vid;
	min = id * tamanio_a / nprocs;
	max = min + tamanio_a / nprocs - 1;

	s = 0;
	for(i = min; i <= max + 1; i++)
		s += a[i];

	barrera();
	for(i = min; i <= max; i++)
		a[i] += s;
}


void init_barrera()
{
	...
}

void destroy_barrera()
{
	...
}

void barrera()
{
	...
}
```

Synopsis

```
int pthread_attr_destroy (pthread_attr_t * attr );
int pthread_attr_getdetachstate (pthread_attr_t * attr , int * detachstate );
int pthread_attr_getinheritsched (pthread_attr_t * attr , int * inherit );
int pthread_attr_getschedparam (pthread_attr_t * attr , struct sched_param * param );
int pthread_attr_getschedpolicy (pthread_attr_t * attr , int * policy );
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_destroy (pthread_cond_t * cond );
int pthread_cond_init (pthread_cond_t * cond , pthread_condattr_t * attr );
int pthread_cond_signal (pthread_cond_t * cond );
int pthread_cond_timedwait (pthread_cond_t * cond , pthread_mutex_t * mutex , struct
timespec * timeout );
int pthread_cond_wait (pthread_cond_t * cond , pthread_mutex_t * mutex );
int pthread_mutex_init (pthread_mutex_t * mutex , pthread_mutexattr_t * attr );
int pthread_mutex_destroy (pthread_mutex_t * mutex );
int pthread_mutex_lock (pthread_mutex_t * mutex );
int pthread_mutex_trylock (pthread_mutex_t * mutex );
int pthread_mutex_unlock (pthread_mutex_t * mutex );
int pthread_once (pthread_once_t * once_control , void (*init_routine) (void) );
pthread_t pthread_self ( );
int pthread_key_create (pthread_key_t * key , void (*destructor) () );
any_t pthread_getspecific (pthread_key_t key );
int pthread_setspecific (pthread_key_t key , any_t value );
int pthread_cleanup_push (void (*fun) (), any_t arg, cleanup_t new );
int pthread_cleanup_pop (int execute );
```