Soluciones a ejercicios de paralelismo y concurrencia

J. Daniel García Sánchez (coordinador)
David Expósito Singh
Javier García Blas
Óscar Pérez Alonso
J. Manuel Pérez Lobato

Arquitectura de Computadores
Grupo ARCOS
Departamento de Informática
Universidad Carlos III de Madrid

1. Problemas de examen de paralelismo

Ejercicio 1 *Examen de junio de 2015.*

Dado el siguiente programa que usa OpenMP:

```
double calcula_pi(double step) {
  int i;
  double x, sum = 0.0;
  #pragma omp parallel for reduction(+: sum) private(x)
  for (i=0;i<1000000;++i) {
    x = (i-0.5) * step;
    sum += 2.0 / (1.0 + x*x);
  }
  return step * sum;
}
```

Escriba un programa OpenMP equivalente que no utilice reducciones.

Solución 1

```
double calcula_pi(double step) {
  int i;
  double x, sum=0.0;
  int nth = omp_getnumthreads();
  double * sumv = malloc(sizeof(double) * nth);
  for (i=0; i<nth; ++i) sumv[i] = 0.0;

  #pragma omp parallel for private(x)
  for (i=0;i<1000000;++i) {
    int id = omp_getthrednum();
    x = (i-0.5) * step;
    sumv[id] += 2.0 / (1.0 + x*x);
  }
  for (i=0; i<nth; ++i) sum+=sumv[i];
  return step * sum;
}
```

Ejercicio 2 *Examen de enero de 2015.*

Dado el siguiente código paralelizado con *OpenMP*, y suponiendo que se disponen de 4 hilos (**export OMP_NUM_THREADS=4**) y que **iter = 16**:

```
#pragma omp parallel for private(j)
for (i = 0; i < iter; ++i) {
  for (j = iter - (i+1); j < iter; ++j) {
    //Función con carga computacional 2s
    compute_iteration(i, j, ...) ;
  }
}
```

Se pide:

1. Rellene en la siguiente tabla una posible asignación de iteraciones de la ejecución del bucle con índice **i** con planificación estática, **schedule(static)**. Indique en la tabla qué hilo realiza cada iteración del bucle (cada valor distinto de **i**) y cuánto tiempo tarda dicha iteración. Calcule además el tiempo aproximado de ejecución por hilo y el tiempo de ejecución total.

# iter	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hilo (ID)															
Tiempo (s)															

2. Rellene en la siguiente tabla una posible asignación de iteraciones ejecutando el bucle con índice **i** con planificación dinámica y *chunk* de 2, **schedule(dynamic, 2)**. Indique en la tabla qué hilo realiza cada iteración del bucle (cada valor distinto de **i**) y cuánto tiempo tarda dicha iteración. Indique además el tiempo aproximado de ejecución por hilo y el tiempo de ejecución total.

# iter	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hilo (ID)															
Tiempo (s)															

3. Justifique cuál de las planificaciones anteriores sería la mejor para un caso genérico (número variable de iteraciones y de hilos).

Solución 2

Apartado 1 En el caso de **static**:

	Iteración															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hilo	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3
Tiempo	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32

Tiempo por hilo:

- Hilo 0: 20 s
- Hilo 1: 52 s
- Hilo 2: 84 s
- Hilo 3: 116 s

Tiempo total: 116s

Apartado 2 En el caso de **dynamic**:

	Iteración															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hilo	0	0	1	1	2	2	3	3	0	0	1	1	2	2	3	3
Tiempo	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32

Tiempo por hilo:

- Hilo 0: 44s
- Hilo 1: 60s
- Hilo 2: 76s
- Hilo 3: 92s

Tiempo total: 92s

Apartado 3 Independientemente del número de iteraciones y de hilos, dado que la carga de trabajo por iteración no es uniforme, se adaptará mejor la planificación dinámica que la estática. Se podría aceptar como respuesta que le mejor planificador es guided.

2. Problemas de examen de concurrencia

Ejercicio 3 *Examen de junio de 2015.*

Dada la siguiente definición de pila libre de cerrojos:

```
template<typename T>
class stack {
private:
  struct node {
    std::shared_ptr<T> data;
    node* next;
    node(T const& data_):data(new T(data_)){}
  };
  std::atomic<node*> head;
public:
  void push(T const& data);
  std::shared_ptr<T> pop()
};
```

1. Proporcione una implementación libre de cerrojos de las funciones **push** y **pop**.
   - **NOTA**: Ignore el problema de los goteos de memoria.
2. Explique brevemente como se podría evitar el problema del goteo de memoria asociado a la solución del apartado anterior.

Solución 3

Apartado 1 Una posible implementación que ignora los goteos de memoria, podría ser:

```
template <typename T>
void push(T const& data) {
        node* const new_node=new node{data};
        new_node->next=head.load();
        while(!head.compare_exchange_weak(new_node->next,new_node));
}

template <typename T>
std::shared_ptr<T> pop() {
        node* old_head=head.load();
        while(old_head && !head.compare_exchange_weak(old_head,old_head->next));
        return old_head ? old_head->data : std::shared_ptr<T>();
}
```

Apartado 2 Se podría añadir un contador atómico con el número hilos que están realizando un **pop** y permitir la eliminación cuando no hay hilos en haciendo **pop**.

Ejercicio 4 *Examen de enero de 2015.*

Sea el siguiente código programado con atómicos. En el punto **A**, **head** contiene un **8** y se intenta insertar un **9**, por lo que se imprimirá la salida *"8 8 9"*. Si otro hilo intenta insertar un **10** de forma concurrente, indique qué datos se imprimirán por pantalla si se ejecuta la parte **B** (a partir de la sentencia de la línea **16**), y qué datos si se llega a ejecutar la parte **C** (a partir de la sentencia de la línea **25**).

```
struct node {
  std::shared_ptr<T> data;
  node* next;
  node(T const& data_):data(new T(data_)), next(nullptr) {}
};

std::atomic<node*> head;

void push(T const& data) {
  node* const new_node=new node(data);
  new_node->next=head.load();

  //A
  std::cout << *(head.load()->data) << " "; // 8
  std::cout << *(new_node->next->data) << " "; // 8
  std::cout << *(new_node->data) << std::endl; // 9

  if (head.compare_exchange_strong(new_node->next,new_node)) {
    //B
    std::cout << *(head.load()->data) << " ";
    std::cout << *(new_node->next->data) << " ";
    std::cout << *(new_node->data) << std::endl;
  }
  else {
    //C
    std::cout << *(head.load()->data) << " ";
    std::cout << *(new_node->next->data) << " ";
    std::cout << *(new_node->data) << std::endl;
  }
}
```

Solución 4

En el caso B, el mensaje por pantalla es:

9 8 9

En el caso C, le mensaje por pantalla es:

10 10 9

No es posible ninguna otra solución, ya que, **compare_exchange_strong** no admite fallos espurios

Ejercicio 5 *Examen de enero de 2014.*

Sea la siguiente función:

```
std::mutex m; // mutex global
int contador; // contador global
void f() {
  m.lock();
  ++contador;
  m.unlock();
}
```

Se desea sustituir la variable global m y evitar posibles llamadas al sistema operativo, pero al mismo tiempo se desea garantizar la exclusión mutua en el incremento del contador.
Se pide:

1. Proponga y codifique una solución que ofrezca consistencia secuencial y no implique llamadas al sistema.
2. Proponga y codifique una solución que ofrezca consistencia de adquisición-liberación.
3. Proponga y codifique una solución que ofrezca consistencia de adquisición-liberación y sea válida en el caso en que el contador pase a ser una variable de tipo double.

Solución 5

Apartado 1 Una posible solución con consistencia secuencial sería:

```
std::atomic<int> contador; // contador global
void f() {
  ++contador;
}
```

Apartado 2 Una posible solución con consistencia de adquisición-liberación sería:

```
std::atomic<int> contador; // contador global
void f() {
  contador.fetch_add(1, std::memory_order_acq_rel);
}
```

Apartado 3 En el caso de valores en coma flotante no se pueden usar tipos atómicos directamente. En este caso se puede conseguir un efecto similar mediante el uso de un *spin-lock*.

```
std::atomic_flag spin = ATOMIC_FLAG_INIT;
double contador; // contador global
void f() {
  while (spin.test_and_set(std::memory_order_acquire)) {}
  contador += 1.0;
  spin.clear(std::memory_order_release);
}
```