Solutions to exercises on parallelism and concurrency

J. Daniel García Sánchez (coordinator)
David Expósito Singh
Javier García Blas

Computer Architecture
ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid

1. Exam exercises related to parallelism

Exercise 1 *(June 2015)*

Given the following program parallelized with OpenMP:

```
double calculate_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;
}
```

Write an equivalent version of the program without the reduction annotation.

Solution 1

```
double calculate_pi(double step) {
  int i;
  double x, sum=0.0;
  int nth = omp_get_num_threads();
  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_get_thred_num();
    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;
}
```

Exercise 2 *(January 2015)*

Given the following code parallelized with *OpenMP* and assuming that we count with 4 threads (**export OMP_NUM_THREADS=4**) and **iter = 16**:

```
#pragma omp parallel for private(j)
for (i = 0; i < iter; ++i) {
  for (j = iter - (i+1); j < iter; ++j) {
    //This function has a computing time of 2s
    compute_iteration(i, j, ...) ;
  }
}
```

State:

1. Fill out the following table with a possible allocation of iterations of the loop with (index **i**) with static scheduling, **schedule (static)**. Indicate in the table which thread performs each iteration of the loop (each value other than **i**) and how long that iteration takes. Also calculate the approximate execution time per thread and the total execution time.

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

2. Fill in the following table with a possible allocation of iterations by running the loop (with index **i**) with dynamic scheduling and *chunk* 2, **schedule (dynamic, 2)**. Indicate in the table which thread performs each iteration of the loop (each value other than **i**) and how long that iteration takes. Also indicate the approximate execution time per thread and the total execution time.

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

3. Justify which of the previous schedules would be best for a generic case (variable number of iterations and threads).

Solution 2

Point 1 In case of a **static** scheduling:

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

Time per thread:

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

Total time: 116s

Point 2 In case of a **dynamic** scheduling:

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

Time per thread:

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

Total time: 92s

Point 3 Regardless of the number of iterations and threads, since the workload per iteration is not uniform, it will be better adapted the **dynamic** scheduling than **static** scheduling. It could be accepted in response that the best scheduler is **guided**.

2. Exam exercises related to concurrency

Exercise 3 *(June 2015)*

Given the following definition of a lock-free stack:

```
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. Provide a lock-free implementation for both **push** and **pop** functions.
   - **NOTE**: Ignore the problem related to memory leaks.
2. Briefly explain how you could avoid the memory leak problem associated with your solution.

Solution 3

Point 1 A possible implementation that ignores memory leaks might be:

```
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>();
}
```

Point 2 You could add an atom counter inicializated with the number of threads that are performing a **pop** and allow deletion when there are no threads that perform **pop**.

Exercise 4 *(January 2015)*

Let the following code be programmed with atomics. At point **A**, **head** contains the value **8** and an attempt is made to insert a value **9**. If another thread tries to insert a value **10** concurrently, indicate which data will be printed on the screen if part **B** is executed (line **16**). Which data if you get to execute the part **C** (line **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;
  }
}
```

Solution 4

In case B, the message in the screen is:

9 8 9

In case C, the message in the screen is:

10 10 9

No other solution is possible, since **compare_exchange_strong** does not support spurious faults.

Exercise 5 *(January 2014)*

Given the following function:

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

We want to replace the global variable *m* and avoid possible system calls, and at the same time, it is desired to ensure mutual exclusion in the *counter* variable increment.

State:

1. Propose and implement a solution that offers sequential consistency and does not involve system calls.
2. Propose and implement a solution that offers release-acquire consistency.
3. Propose and implement a solution that offers release-acquire consistency and it is valid in case of the *counter* variable becomes a double-precision float-point number.

Solution 5

Point 1 A possible solution applying sequential consistency would be:

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

Point 2 A possible solution applying release-acquire consistency would be:

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

Point 3 In the case of floating-point variables, atomic types can not be used directly. In this case, a similar effect can be achieved by using a *spin-lock* approach.

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