# Exercices – Course 2.37.1 – Semantics, languages and algorithms for multicore programming

February 21, 2017

### Exercise 1. Variations on task pipelines

We consider the following C function, computing $\mathbf{z} = a\mathbf{x} + \mathbf{y}$, also known as the SAXPY numerical kernel (S for single precision floating point numbers):

```
void saxpy(int n, float a, const float x[n], const float y[n], float z[n])
{
  for (int i=0; i<n; i++) {
    z[i] = a * x[i] + y[i];
  }
}
```

All iterations are independent and can be run in parallel. A Cilk version of this function running each iteration as a concurrent task is shown below:

```
cilk float saxipyi(float a, float xi, float yi)
{
  return a * xi + yi;
}

cilk void saxpy(int n, float a, const float x[n], const float y[n], float z[n]) {
  for (int i=0; i<n; i++) {
    z[i] = spawn saxipyi(a, x[i], y[i]);
  }
  sync;
}
```

**Question 1**.

The absence of dependences across loop iterations allows to exploit data parallelism in the program.

What is the maximal degree of data parallelism of the SAXPY kernel, i.e., the maximal number of operations that can run in parallel, exploiting data parallelism alone?

**Answer:** $n$, the number of iterations in the loop.

**Question 2**.

The "work-first" policy of Cilk consists in the sequential execution of the `spawn`-qualified function call, while the continuation of the parent function is pushed to a deque for parallel execution.

When running `saxpy` on $p$ processors, what is the maximal memory usage of the stack frames of concurrently running tasks?

**Answer:** Let $s$ be the maximum of the size of one stack frame of the `saxipyi` function, and the size of one stack frame of the continuation of the `saxpy` function.
Tasks are pushed sequentially, so there can be at most one task in the deques at any given time, and at most one continuation frame active. While the `saxipyi` function is running, another processor may steal the continuation task and push the continuation of the next iteration. At most $p$ such steals may occur concurrently, meaning that at most $p$ instances of the `saxipyi` stack frame may be simultaneously active.
The maximal memory usage is $(p + 2) \times s$, counting the frame of the `saxpy` function and the frame of the continuation task (they may co-exist for a short time, but $p + 1$ also an acceptable answer).

**Question 3**.

Same question in the case of the "help-first" policy, where the `spawn`-qualified function is pushed to a deque while the the continuation of the parent function is executed sequentially.

Discuss which policy is most suitable for this program, depending on $p$ and $n$.

**Answer:** There can be up to $n$ tasks associated with instances of `saxipyi` in the deques or running on the processors at any given time. In addition, the frame of the `saxpy` function remains in the stack at all times.
The maximal memory usage is $(n+1) \times s$.
In general, $p$ is much lower than $n$. The work-first policy is much more economical.

**Question 4**.

Cilk only allows to express fork-join parallelism. To exploit pipeline parallelism, we add a syntax for *futures* to the language.

A future `p` of type `T` is declared `future T p`. It is a reference, holding the future value produced by some concurrent task(s).

The `void set(future T *p, T v)` function defines the future `*p` to the value `v`.

The `T get(future T *p)` function waits for the availability of the data held in the future `*p`, and returns its value.

This is best illustrated on an example:

```
cilk void producer(future int *p, int i)
{
  set(p, i);
}

cilk void consumer(future int *p, int *q)
{
  *q = get(p) + 42;
}

void main()
{
  future int x[10];
  int y[10];
  for (int i=0; i<10; i++) {
    spawn producer(&x[i], i);
    spawn consumer(&x[i], &y[i]);
  }
  sync;
  // do whatever with y
}
```

Each loop iteration creates two tasks: the producer task writes into a future private to this particular iteration, and the consumer task reads from it using `get()` to wait for the availability of the data.

What is the value of `y[9]` after the `sync` keyword?

Assuming `y` is intialized to 0, what are the possible values of `y[9]` if it was read after the loop and before the `sync` keyword?

**Answer:** 0 and 51.

**Question 5**.

Name a major drawback of this low-level programming interface with explicit assignments to futures, compared to the C++ (or F#) futures studied during the course.

It also has some advantages, can you name one?

**Answer:** There is no guarantee of determinism or even race freedom, as futures may be assigned multiple times, even concurrently.
But explicit management of futures allows to reuse memory and implement in-place operations, avoiding the garbage collection performance overhead of futures.

**Question 6**.

In this question, we assume single-assignment operations on futures, i.e., `set()` is not called more than once on a given future object.

The Cilk memory model is called DAG consistency. Memory events on a path induced by `spawn` and `sync` are totally ordered.

Propose two extensions of DAG consistency for Cilk programs with futures. Both should enforce coherence of `set()` and `get()` for a given future, but the second should be weaker than the first in the way memory events on unrelated shared variables are propagated across `set()` and `get()`.

Give an example of a compiler optimization that is allowed for the second model but not for the first one.

**Answer:** `set()`/`get()` edges can be added to the DAG of `spawn`s and `sync`s.

- In the first model, they enforce a total ordering of memory events, just like the other edges.
- In the second model, they may only enforce ordering of memory events on the future object itself (coherence).

An assignment to a shared variable may not be moved across a `set()` operation with the first model, but it is allowed with the second one. On the other hand, in the second model, the happens-before relation across shared variables and futures will need extra synchronizations.

**Question 7**.

The SAXPY kernel exhibits some potential for pipelined execution. Write a parallel version where each iteration runs as a pair of Cilk tasks communicating through a future.

**Answer:**

```
cilk void ax(float a, float xi, future float *p)
{
  set(p, a * xi);
}

cilk void py(float yi, future float *p, float *zi)
{
  *zi = get(p) + yi;
}

cilk void saxpy_split(int n, float a, const float x[n], const float y[n], float z[n])
{
  for (int i=0; i<n; i++) {
    spawn ax(a, x[i], &p[i]);
    spawn py(y[i], &p[i], &z[i]);
  }
  sync;
}
```

**Question 8**.

What is the maximal degree of task parallelism of the SAXPY kernel, i.e., the maximal number of operations that can run in parallel, irrespectively of the iteration and task they are issued from?

**Answer:** It is still $n$. No improvement compared to data parallelism only.

**Question 9**.

What is the main performance benefit of combined pipeline and data parallelism over data parallelism only?

**Answer:** It is less demanding on memory bandwidth when hiding memory latency with concurrent tasks: task-to-task communications may use on-chip resources whereas data parallelism abuses locality and stresses off-chip memory interfaces.

## Exercise 2. Fibonacci again

We would like to explore a bit further the Cilk-based parallelization of the Fibonacci function studied during the course.

```
cilk int fib(int n)
{
  if (n < 2)
    return n;
  else {
    int x, y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}
```

Note that the second `spawn` does not add any parallelism because it is immediately followed by a `sync`. We will keep it for the sake of the illustration and exercise anyway.

The "work-first" policy of Cilk consists in the sequential execution of the `spawn`-qualified function call, while the continuation of the parent function is pushed to a deque for parallel execution.

**Question 10**.

What is the maximal size of the deque for a given thread, as a function of $n$, running the program with the work-first policy of Cilk?

**Answer:** Let $s$ be the maximum of the size of one frame of the `fib` function or one of the continuations generated by the Cilk compiler.
In the absence of steals, the execution on each processor follows a left-handed, depth-first exploration of the call tree (corresponding to a strict/eager evaluation strategy). While a processor explores its sub-tree, the frame of the second call is pushed to the deque, hence the deque size is bounded by the number of recursive calls, i.e., $n \times s$.
When a steal occur, the deque of the thief is initially empty, then grows with the exploration of a new sub-tree. At all times, the deque of a given processor is bounded by $n \times s$.

**Question 11**.

Assuming the program runs with the opposite, help-first policy where spawn pushes its coroutine argument rather than pushing its continuation, what would be the maximal size of the deque for a given thread?

**Answer:** In the absence of steals, the execution on each processor still follows a depth-first exploration of the call tree, but this time it corresponds to to a lazy evaluation strategy: the task running the `fib` function pushes its two children, then suspends on the `sync` operation. While suspended, the work-stealing scheduler takes the bottom-most task from the processors's deque and runs it. Each task will thus push two children tasks before suspending, and the depth-first exploration will be a right-handed one. Counting the suspended tasks and the first children pushed on the deque, the maximal number of frames in a given processor's deque is $(2 \times n + 1) \times s$.
When a steal occur, the stack of the thief is initially empty, then grows with the exploration of a new sub-tree. At all times, the stack of a given processor is bounded by $(2 \times n + 1) \times s$.
Note that in the absence of a `sync` operation, the size of an individual deque may grow exponentially in $n$.

**Question 12**.

Transform the fib program to use futures instead of Cilk's primitives. You may use any existing language syntax, or pseudo-code at your own taste, as long as future values are distinctively typed, created, and bound (`get()` operation).

Does this version expose more parallelism? Does it impact the number of synchronisations or the load balance of the worker threads executing asynchronous tasks?

**Answer:** Let us choose a low-level approach to the implementation of futures in Cilk, using *future* objects.
A future `p` of type `T` is declared `future T p`. It is a reference, holding the future value produced by some concurrent task(s).
The `void set(future T *p, T v)` function defines the future `*p` to the value `v`.
The `T get(future T *p)` function waits for the availability of the data held in the future `*p`, and returns its value.
Here is a possible implementation of `fib` with these futures:

```
cilk void future_fib(int n, future int *p)
{
  if (n < 2)
    set(p, n);
  else {
    future int p1, p2;
    spawn future_fib(n-1, &p1);
    spawn future_fib(n-2, &p2);
    set(p, get(&p1)+get(&p2));
  }
}

int fib(int n) {
  future int p;
  future_fib(n, &p);
  return get(&p);
}
```

### Exercise 3. Parallel histogram

We would like to parallelize the computation of the number of pixels of a given intensity in a grayscale image. Each pixel is represented by byte: `unsigned char img[m][n];`. The result is an array `int hist[NB]` where `hist[`$i$`]` records the number of pixels of intensity $i$ in the image and `NB` is the number of grayscale levels, here 256.

More generally, the histogram array is indexed into so-called *buckets*, which can be keys in a sort algorithms, values of a physical simulation, etc.

**Question 13**.

A very naive parallel implementation would make use of two nested OpenMP parallel for loops with a critical section on the incrementation of `hist[img[`$i$`][`$j$`]]`. A more reasonable one would use a single atomic fetch-and-add operation. Sketch these 2 versions as OpenMP pseudo-code.

**Answer:** Critical section:

```
  void histogram()
  {
#pragma omp parallel for collapse (2)
    // Data parallel work-sharing over a 2D image
    for (int i = 0; i < m; i++)
      for (int j = 0; j < n; j++)
#pragma omp critical
        {
          hist[img[i][j]]++;
        }
  }
```

It is possible to isolate the incrementation and then to replace `critical` with `atomic` above to avoid the global lock, and even to avoid any lock, resorting only to a single atomic read-modify-write operation:

```
  void histogram()
  {
    //
#pragma omp parallel for collapse (2)
    // Data parallel work-sharing over a 2D image
    for (int i = 0; i < m; i++)
      for (int j = 0; j < n; j++)
      {
        int b = img[i][j];
#pragma omp atomic
        hist[b]++;
      }
  }
```

**Question 14**.

What is the main weakness of these versions with a lock-based critical section or compare-and-swap operation?

**Answer:** Contention on the global lock completely serializes the execution in the naive version. Contention on the atomic incrementation will be much more manageable for low thread counts, but on larger machines with more processors the impact on cache coherence and contention will grow, reducing scalability.

**Question 15**.

Propose an optimization taking advantage of the associativity and commutativity of the addition, where partial incrementations occur sequentially on each processor, and a final reconciliation step reduces the partial sums into the resulting histogram array. No need to implement this version.

**Answer:** Since the image is thought to be much larger than the `hist` array (256 elements), it is profitable to partition the image into blocks and run partial histogram computations into private arrays for each block, then to complete the full histogram in one sequential traversal of all these private arrays (or even to parallelize this traversal with the above scheme).

**Question 16**.

The previous approach performs well when the number of buckets (here, grayscale levels) is low. But it is not the case when the number of buckets can be larger than the data being sampled itself. This is often the case for variants of the histogram algorithm where buckets represent sorting keys or (intervals of) physical simulation values over discretized domains. When dealing with large histogram arrays, it is sometimes possible to parallelize the program in iterative waves. The sketch of the algorithm is as follows, computing the histogram of some input array of integers `data[n]` partitioned over the different processors, and using additional shared arrays `int elected[n]` initialized to true, `int winner[NB]`, and a shared boolean variable `live`.

1. Initialize the `winner` array to 0 and set `live` to false.
2. Each processor traverses its partition of the data sequentially, assigning $i$ to `winner[data[`$i$`]]` and setting `live` to true for each index $i$ of the partition such that `elected[`$i$`]` is true. Intuitively, the `winner` array allows to elect the data element that will contribute its increment to the corresponding histogram bucket in the current wave, and the `elected` array tracks the elements that have won the race and should be ignored in the next race.
3. Synchronization barrier (wait for all processors to complete).
4. Traverse the `winner` array in parallel, for each $i$ such as $j$ =`winner[`$i$`]` is not 0, increment `hist[data[`$i$`]]` then set `elected[`$i$`]` to false.
5. Synchronization barrier (wait for all processors to complete).
6. Iterate if `live` is true.

(The real algorithm is slightly more complex. Since multiple elements in a given partition may fall in the same bucket, it is inefficient to increment the shared histogram array one by one. But fixing this inefficiency does not impact the concurrency issues considered in the exercise.)

There are 2 sources of data races in this algorithm. Which ones? What is specific about these races?

**Answer:** The races are associated with the concurrent marking of the shared `live` variable to detect termination, and the selection of the winner for each bucket. Interestingly, both are *benign* data races.

**Question 17**.

Assuming a pthread C11 implementation of the algorithm and synchronization barrier, describe the memory model issues and low-level atomics implementation to deal with these races correctly.

Note: you may assume the synchronization barrier takes two steps, first each thread decrements a shared counter initially set to the total number of threads, and then it waits until the counter reaches 0.

**Answer:** Low-level atomics are needed for the stores to the shared variable `live` and array elements `winner[`$i$`]`. The good news is that it is sufficient to implement these atomic stores with a *relaxed* semantics. Indeed, the barrier implementation will need to make the atomic decrement of the shared counter a *release* operation and the check for 0 in the waiting loop an *acquire* operation. This is sufficient to guarantee that any *relaxed* store preceding the barrier in program order happens before the subsequent phase of the computation following the barrier.