Exercise 1. Variations on task pipelines

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

```c
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:

**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?

**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?

**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 \( n \).

**Question 4.**

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

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

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

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

This is best illustrated on an example:

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

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

  void main() {
    promise 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 promise 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 initialized to 0, what are the possible values of y[0] if it was read after the loop and
before the sync keyword?

**Question 5.**
Name a major drawback of this low-level programming interface with explicit assignments to promises,
compared to the C++ (or F#) futures studied during the course.
It also has some advantages, can you name one?

**Question 6.**
In this question, we assume single-assignment operations on promises, i.e., set() is not called more
than once on a given promise 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 promises. Both should enforce
coherence of set() and get() for a given promise, 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.

**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.

**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?

**Question 9.**
What is the main performance benefit of combined pipeline and data parallelism over data parallelism
only?

**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?

**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?

**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?

**Exercise 3. Smith-Waterman**

**Question 13.**
The SmithWaterman algorithm is a well-known “dynamic programming” algorithm for performing local sequence alignment; that is, for determining similar regions between two nucleotide or protein sequences. Instead of looking at the total sequence, the SmithWaterman algorithm compares segments of all possible lengths and optimizes the similarity measure.

A matrix \( H \) is built as follows:

\[
H(i, 0) = 0, \quad 0 \leq i \leq m \\
H(0, j) = 0, \quad 0 \leq j \leq n \\
\text{if } a_i = b_j \text{ then } w(a_i, b_j) = W_{\text{match}} \text{ or if } a_i \neq b_j \text{ then } w(a_i, b_j) = W_{\text{mismatch}}
\]

\[
H(i, j) = \max \begin{cases} 
0 & \text{Match/Mismatch} \\
H(i-1, j-1) + w(a_i, b_j) & \text{Deletion} \\
H(i-1, j) + w(a_i, -) & \text{Insertion} \\
H(i, j-1) + w(-, b_j) & \text{Insertion}
\end{cases}, \quad 1 \leq i \leq m, 1 \leq j \leq n
\]

Where:
\( a, b = \text{Strings over the Alphabet } \Sigma \)
\( m = \text{length}(a) \)
\( n = \text{length}(b) \)
\( H(i, j) \) is the maximum Similarity-Score between a suffix of \( a[1...i] \) and a suffix of \( b[1...j] \)
\( w(c, d), \quad c, d \in \Sigma \cup \{\cdot\} \)
\( \cdot \) is the gap-scoring scheme

Write a sequential implementation of this algorithm.

**Question 14.**
Write a Cilk implementation of the Smith-Waterman algorithm.

**Question 15.**
Write a parallel version of the program with futures.

**Question 16.**
Does this version expose more parallelism? Does it impact the number of synchronisations or help balance the load across the processors?

**Exercise 4. Task pipeline on a linked list**

**Question 17.**
We would like to parallelize the traversal of a linked list of `ints`, applying a function `int slow(int)` to each node of the list, and storing the returned values in any order in a new list.
Write a pseudo-code program using a pthread-like programming interface, where a front-end thread traverses the list, pushing the values into a first FIFO, a collection of threads (the number of threads should be a parameter) pop from the FIFO and apply the \texttt{slow()} function, then insert the results into a new list.

**Question 18.**

Transform the previous program to preserve the ordering of the values returned by each application of the \texttt{slow()} function.