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

February 21, 2018

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?

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 *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?

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?

Question 6.

How would you program this using OpenMP dependent tasks?

Question 7.

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.

Question 8.

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 9.

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 10.

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 11.

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 12.

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 13.

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?

Question 14.

Transform the fib program to use OpenMP fork-join tasks instead of Cilk's primitives, i.e., the taskwait directive for synchronization only.

Question 15.

Transform the fib program to use OpenMP dependent tasks instead of Cilk's primitives, i.e., using output and input clauses for synchronization whenever applicable.

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 16.

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.

Question 17.

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

Question 18.

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.

Question 19.

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?

Question 20.

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.

Exercise 4. Smith-Waterman

Question 21.

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:

$H(i,0) = 0,\ 0 \leq i \leq m$

$H(0,j) = 0,\ 0 \leq j \leq n$

if $a_i = b_j$ then $w(a_i, b_j) = W_{(\text{match})}$ or if $a_i \neq b_j$ then $w(a_i, b_j) = W_{(\text{mismatch})}$

$$H(i,j) = \max \left\{ \begin{array}{cc} 0 & \\ H(i-1,j-1) + \ w(a_i, b_j) & \text{Match/Mismatch} \\ H(i-1,j) + \ w(a_i, -) & \text{Deletion} \\ H(i,j-1) + \ w(-, b_j) & \text{Insertion} \end{array} \right\}, \ 1 \leq i \leq m, 1 \leq j \leq n$$

Where:
a, b = Strings over the Alphabet Σ
$m = \text{length}(a)$
$n = \text{length}(b)$
$H(i,j)$ is the maximum Similarity-Score between a suffix of $a[1...i]$ and a suffix of $b[1...j]$
$w(c,d), \ c,d \in \Sigma \cup \{'\_'\}, '\_'$ is the gap-scoring scheme
Write a sequential implementation of the Smith-Waterman algorithm.

Question 22.
Write a Cilk (pseudo-code) implementation of the Smith-Waterman algorithm.

Question 23.
Write an OpenMP data-parallel (pseudo-code) implementation of the Smith-Waterman algorithm with parallel for loops.

Question 24.
Modify the Cilk implementation to rely on the futures introduced in the second exercise rather than joining tasks with `sync`.

Question 25.
How do these different versions differ in terms of parallelism degree? In terms of the number of synchronization opertions? In terms of load (im)balance across the processors?