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

February 19 2014

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:

Answer:

```
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 still 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 *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 `*p` to the value `v`.

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

This is best illustrated on an example:

```
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 intialized to 0, what are the possible values of `y[0]` 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 promises, 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, as promises may be assigned multiple times, even concurrently.
But explicit management of promises allows to reuse memory and implement in-place operations, saving some performance overhead of futures (garbage collection).

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.

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 a total ordering of memory events on the promise 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.

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, promise float *p) {
  set(p, a * xi);
}

cilk void py(float yi, promise 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 *promises* objects.
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 `*p` to the value `v`.
The `T get(promise T *p)` function waits for the availability of the data held in the promise `*p`, and returns its value.
Here is a possible implementation of `fib` with these promises:

```
cilk void future_fib(int n, promise int *p) {
  if (n < 2)
    set(p, n);
  else {
    promise 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) {
  promise int p;
  future_fib(n, &p);
  return get(&p);
}
```

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:

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

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

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 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?

Answer: The Cilk implementation uses a `sync` barrier to wait for all tasks (i,j) with $i+j \leq k$ before running any task (i',j') with $i'+j' = k$. These barriers are costly because a lot of tasks need to be accounted for, and because of the load imbalance they are likely to induce.
The implementation with futures only synchronises the consumer task (i,j) with its two producer tasks $(i-1,j)$ and $(i,j-1)$. The parallel code is much easier to write and also much more likely to be well balanced using a work-stealing scheduler.

Exercise 4. Task pipeline on a linked list

Question 17.

We would like to parallelize the traversal of a linked list of `int`s, 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 `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 `slow()` function.