CS243 Homework 5

Winter 2021

Due: March 3rd, 2021 at 4:00 pm

Directions:

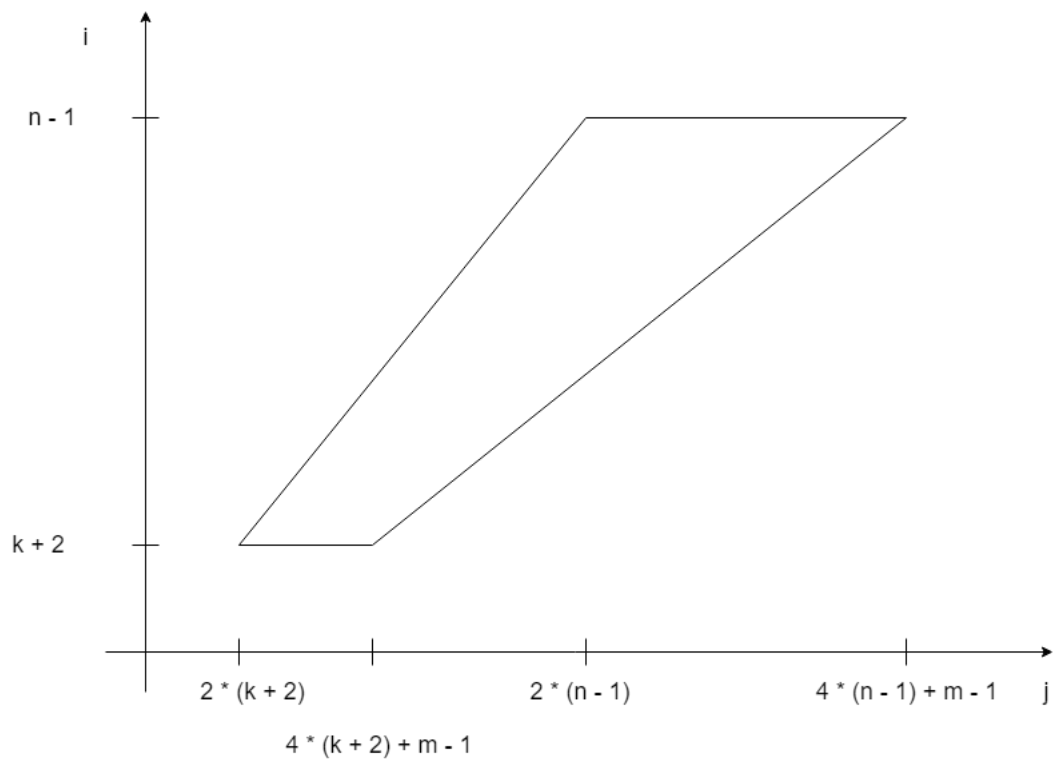
- You may use up to two of your remaining late days for each part of this assignment, for a late deadline of March 5th, 2021 at 4:00 pm (the final deadline for both parts of this assignment).
- This homework includes a Gradiance quiz (please see the website), some written questions, and a programming component; the latter two are to be submitted via Gradescope (as separate submissions). No late days can be applied to the Gradiance quiz.
- The written portion of this assignment (problems 1-3) is an individual assignment. You are allowed to discuss the homework with others, but you must write the solution individually. If you look up any material in the textbook or online, you should cite it appropriately.
- The programming portion of this assignment can be done in pairs. If you are working in a pair, be sure to add your partner to your programming assignment's Gradescope submission when submitting your code. See problem 4 for more instructions.

Problem 1. Dependency Analysis and Parallelization

```
for (i = k + 2; i < n; i++) {
    for (j = 2 * i; j < 4 * i + m; j++) {
        X[i, 4 * j - 2] = X[i, 4 * j + 1] + Y[i, j]
        Y[i - 1, j - 3] = X[i, 2 * j] + Y[i, j]
    }
}
```

Assume $0 \leq k \leq m << n$ and that `X` and `Y` are distinct (non-overlapping) arrays.

1. Draw the iteration space for this loop.



2. What are the data dependencies in this loop? Hint: a data dependency is something of the form "read/write A[i], read/write B[j]".

   write `X[i, 4 * j - 2]`, read `X[i, 4 * j + 1]` (optional since 4 * j - 2 cannot equal 4 * j + 1 by modular arithmetic)
   write `X[i, 4 * j - 2]`, read `X[i, 2 * j]`
   write `Y[i - 1, j - 3]`, read `Y[i, j]`

   write `Y[i - 1, j - 3]`, write `Y[i - 1, j - 3]` (optional)
   write `X[i, 4 * j - 2]`, write `X[i, 4 * j - 2]` (optional)

3. Formulate the data dependence tests for the given loop nest.

   Bounds equations:
   $k + 2 \leq i < n$
   $k + 2 \leq i' < n$
   $2i \leq j < 4i + m$
   $2i' \leq j' < 4i' + m$

   (a) write `Y[i - 1, j - 3]`, read `Y[i, j]`:
   $i - 1 = i' \wedge j - 3 = j'$

   (b) write `X[i, 4 * j - 2]`, read `X[i, 2j]`
   $i = i' \wedge 4j - 2 = 2j'$

   Make sure the indices aren't the same:
   $[i, j] \neq [i', j']$

   For each of the inequalities in (a)-(b) combined with the bounds conditions and unique indices test, if the equations have integral solutions, then there is a data dependence.

4. Is the loop nest 1-d parallelizable or 2-d parallelizable without loop transformations (or not parallelizable at all)? You do not need to show the exact solutions to the equations, but justify your answer.

   Suppose $k = 0$ and $m = 0$.

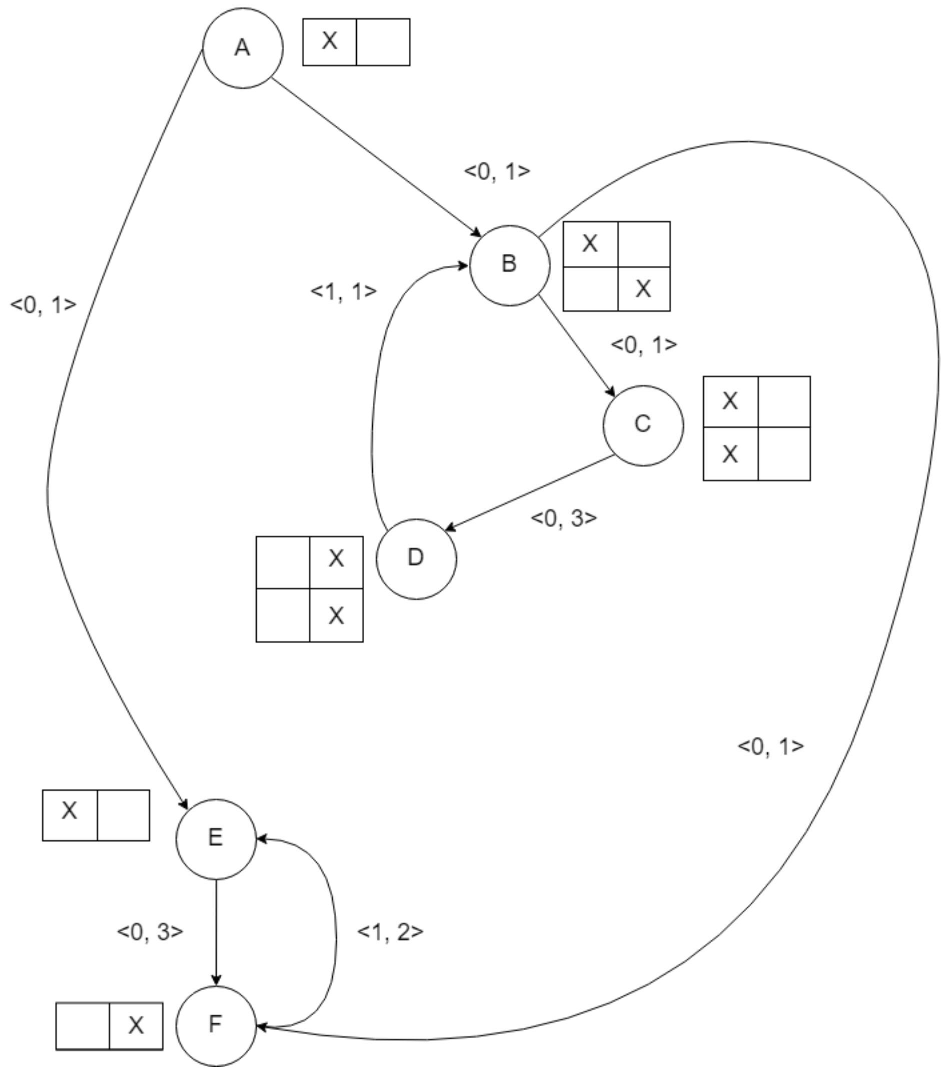
   The inner loop is not parallelizable, as the `X[]` writes and reads can overlap for fixed `i` indices. As an example, `(i = 5, j = 10)` will write to `X[5, 38]`, while `(i = 5, j = 19)` will read from `X[5, 38]`.

   The outer loop is not parallelizable either, as `(i = 11, j = 25)` will write to `Y[10, 22]`, while `(i = 10, j = 22)` will read from `Y[10, 22]`.

   Therefore, this loop nest is not parallelizable at all.

Problem 2. Software Pipelining

Consider the following dependence graph for a single iteration of a loop, with resource constraints:



1. What is the bound on the initiation interval T according to the precedence and resource constraints for this program?

   5 due to data dependence in cycle. Max (cycle length/ iteration diff) = 5/1 = 5
   5 due to resource constraints as well (left column resource).

2. What is the minimum initiation interval? Show a modulo reservation table for an optimal software pipelined schedule. Also show the code and schedule for an iteration in the source loop.

   The minimum initiation interval is 5. Modulo reservation table:

A	
E	D
B	D
C	B
C	F

Single iteration: A E B C F nop D

3. Can the scheduling algorithm described in class produce the optimal schedule for this loop? If not, show the modulo reservation table and code schedule generated by the algorithm.

The scheduling algorithm described in class will backtrack only within strongly connected components, and schedule greedily across them. Therefore it will not produce an optimal schedule: it will pack the $\{B, C, D\}$ SCC immediately after A (in the first available spot), eliminating the ability interleave the $\{E, F\}$ SCC in-between.

A	D
B	D
C	B
C	
E	

Here, we are stuck, since we cannot schedule F 3 clocks after E.

However, the algorithm will be able to schedule instructions with an initiation interval of 6. Modulo reservation table:

A	D
B	F
C	B
C	
E	
	D

Single iteration: A B C nop nop D nop nop nop nop E nop nop F

Problem 3. Affine Transforms

Apply affine transform and loop fusion to find the largest degree of outermost loop parallelism. Show the transformed code, marking the loops that are parallelizable.

```
for (i = 1; i <= N; i++) {
  for (j = 1; j <= N; j++) {
    A[i,j] = A[i-1,j] + X[i,j];
  }
}
for (i = 2; i <= N; i++) {
  for (j = i+1; j <= N; j++) {
     for (k = 1; k <= N; k++) {
        B[i,k] = B[i-1,k] + Y[j];
    }
  }
}
for (i = 1; i <= N; i++) {
   C[i] = A[1,i] + B[1,i];
}
```

Solution:

```
for (pj = 1; pj <= N; pj++) { // parallelizable
    for (pi = 1; pi <= N; pi++) {
        A[pi, pj] = A[pi-1,pj] + X[pi,pj];
        if (2 <= pi) {
            for (j = pi+1; j <= N; j++) {
                B[pi,pj] = B[pi-1,pj] + Y[j];
            }
        }
    }
    C[pj] = A[1,pj] + B[1,pj];
}
```

There is only one degree of outermost loop parallelism.

Problem 4. Programming

Directions:

- Download the starter code from the course website, complete the task (see “Your Task” below), and submit your code and write-up using Gradescope.
- This assignment is graded by code inspection, so please make sure your code is readable and any design decisions are documented.

Optimizing Numerical Code for Parallelism and Locality

The goal of this homework is for you to learn how to write parallel code, and how to optimize parallelism in existing numeric code.

Your task will be to take some existing, serial code, and write the corresponding parallel code, to achieve the best performance.

Setting up the environment

To make the best use of real machines, this homework is written in C++.

If you use Stanford rice: If you do not have access to a C++ coding environment, you should use the one in Stanford rice[1]. In that case, just unzip the starter code; use `make` to build the compiled program. All paths are already set up.

Note: rice and myth machines are similar, with rice slightly newer and more powerful. They have the same C++ compiler version, and they share the home directory. You should use rice for this homework, because myth is not appropriate for intensive CPU use.

Note 2: not all rice or myth machines are identical. When you compile, you are optimizing for the specific machine your are compiling in, which can make the code not portable. If you see an “Illegal Instruction” error, simply rebuild (`make clean` followed by `make`).

If you set up your own platform: You must have a GCC-compatible C++ compiler (such as GCC itself, or Clang), which must support C++11 and OpenMP. Any modern C++ compiler, including the one in Ubuntu 16.04 LTS will do. Grading will occur on rice; mind any divergence in C++ standard or compiler.

Parallel C++ code

The parallel C++ code in this handout makes use of OpenMP. This allows us to write code like this:

```
#pragma omp parallel for schedule(static,1)
for (int i = 0; i < N; i++) {
```

[1]See https://srcc.stanford.edu/farmshare2 for how to access the rice machines.

```
    ...
}
```

What this statement means is that the entire loop will be 1-d parallelized: the iteration space of the outermost loop will be split among the available CPUs and then the result will be run in parallel. After the parallel loop is done, execution is *joined* and resumes in the main thread. This way we do not need to worry about threading or locking, and we just need to write the outermost loop parallel code, as discussed in class.

The directive `schedule(static,1)` indicates that each thread will be assigned one iteration of the loop, in round-robin fashion. This allows us to use iteration variable as the processor id, for example to synchronize on it in pipeline parallel code. This directive is not the default: if you omit it, you let the implementation choose the best schedule.

We can also write 2-d parallel code, using:

```
#pragma omp parallel for collapse(2) schedule(static,1)
for (int i = 0; i < N; i++) {
  for (int j = 0; j < M; j++) {
    ...
  }
}
```

And similarly, 3-d parallel code, 4-d, etc.

On the other hand, OpenMP can be used only if the outermost loops are perfectly nested, parallel and use an integer induction variable with loop-invariant bounds. In this assignment, you will therefore need to rewrite your loops to satisfy these constraints - or pay the penalty of not having properly parallelized code.

For further details of OpenMP, see its specification at http://www.openmp.org/specifications/. You do not need to understand all of OpenMP for this assignment, though.

1 Basics of Writing Parallel Code

To start, we will consider Matrix Multiplication, as introduced in class. You can find the full code for this task in `matmul.cpp` in the starter code.

The starter code introduces the `Matrix` class, a row-major, packed, dense Matrix class of arbitrary types (mostly `float` or `double`). It also introduces the skeleton code to generate random matrices and evaluate the performance.

You run the starter code as `./matmul` r_1 c_1 c_2. This causes a matrix multiplication between two matrices, of size $r_1 \times c_1$ and $c_1 \times c_2$.

The output looks like this:

```
Iteration 1: 2272 us
Iteration 2: 2233 us
Iteration 3: 2238 us
Iteration 4: 1304 us
Iteration 5: 1302 us
Iteration 6: 1278 us
...
Avg time: 1301 us
Stddev: ±242 us
```

The starter code includes three implementations of matrix multiplication: serial naive, parallel naive, and parallel blocked. You can choose which one is invoked by modifying the starter code.

1. Test matrix multiplication between two matrices both of size 1000 by 1000. What is the average time? Which is the fastest algorithm?
2. Test matrix multiplication between matrix A of size 1000 by 10, and matrix B of size 10 by 1000. What is the average time? Which is the fastest algorithm?

2 Pipelined Parallelism

In this problem, you will consider the Successive Over-Relaxation code introduced in class, and implement a pipelined parallel version of it. You can refer to Lecture 11 for the definition.

The starter code includes a serial version of Successive Over-Relaxation. You can run it as `./sor` n m, where n and m are the sizes of the matrix to compute on.

Your task is:

1. Write a parallel version of SOR in `sor.cpp`. You can use wavefronting or use pipelining based on semaphores. An example synchronization primtive is included in the starter code.
2. What is the average time to compute SOR with:
   (a) input of size 20000×20000
   (b) input of size 20000×1000
   (c) input of size 1000×20000

3 Kernel Fusing the LSTM

In this task, we will consider how to pair matrix multiplication with component-wise operations, as commonly done by neural networks. As the example, we will consider the Long Short Term Memory Recurrent Neural Network Cell. This is a primitive neural network operation that takes an input x, a state h, a memory cell c, and produces a new state h' and a new memory cell c'. The equations for the LTSM are:

$$
\begin{aligned}
f &= \sigma(hW_f + xU_f + B_f) \\
i &= \sigma(hW_i + xU_i + B_i) \\
o &= \sigma(hW_o + xU_o + B_o) \\
j &= \tanh(hW_c + xU_c + b_c) \\
c' &= f \circ c + i \circ j \\
h' &= o \circ \tanh(c')
\end{aligned}
$$

where σ is the sigmoid function ($\frac{1}{1+\exp(-x)}$), tanh is the hyperbolic tangent, and $\circ$ denotes component-wise multiplication. W, U and B are the learned parameters of the LSTM. W and U are rectangular matrices, and B is a row vector.

f, i and o are called the *gates* of the LSTM, and they control which parts of the previous memory should be preserved (f = forget gate, i = input gate), and which parts of the memory should be exposed to the output state (o = output gate)[2]. j is the value computed from the input, which is written to the memory cell.

In the equation, x is a single data-point vector, but in real-life, and therefore in this assignment, x is a batch of inputs as a matrix, with one input per row. Similarly, the outputs h' and c' will be matrices, one output per row. For efficiency, we will also pack the W's and U's into a single parameter matrix.

The starter code includes a naive implementation of LSTM, in terms of matrix multiplication and component-wise operations. You run the starter code as `./lstm` b x h, where b is the batch size (number of elements in the input), x is the size of the input, and h is the size of (hidden) state and memory cell. You should observe that given those two parameters, all matrices have known size.

Your task is:

1. Write an efficient serial implementation of the LSTM in `lstm.cpp`. Efficient means that will make the best use of caches, registers and instruction level parallelism.

2. Transform your serial code into an efficient parallel implementation.

3. What is the average time to compute the LSTM with:

   (a) batch of size 500, input of size 2000 and output of size 2000

   (b) batch of size 500, input of size 100 and output of size 5000

4. Does the parallel implementation scale with the number of processors? Draw a weak scaling plot (average time vs number of processors) for an input and output of size 5000,

[2] The gates of an LSTM use sigmoid activation. Because sigmoid saturates to 0 or 1 quickly when the input is away from zero, the gates look like binary vectors. This does not matter to us, though: to us, they are just floating point vectors.

starting at 1 and ending with the maximum number of processors in a rice machine (16).

To test with different number of processors, run your code as `OMP_NUM_THREAD=`x `./lstm ...`, where x is the desired number of threads.

To draw the plot, you can use any plotting library (matplotlib, R, or D3), or an application such as Google Spreadsheet. Do not draw the plot by hand. Please also label your axes, especially if you use a non-standard scale.

In this task, you must write the most efficient code possible, applying all the loop transformations for locality and parallelism discussed in class. You do not need to explain the solution to any data-dependency equation, if you use them. On the other hand, your code will be graded by inspection, so you should document all design decisions you make.

Submission

To submit, run `make submission`. Download `submission.zip` and upload to Gradescope. Only one submission is required per pair.

You must not modify the submitted code, outside of the areas marked with "YOUR CODE HERE". You can modify the rest of the code for debugging, but please revert your changes before submission.

In the same folder, you must include a `writeup.pdf` with the measurements and the answers to the written questions, and a PNG with the plot for part 2.4. Only one writeup is required per pair.

Hints

- As always, start early. This homework uses a different environment than joeq, and it might take some time to get adjusted.
- The compiler is your friend. It is often safe to assume that certain method calls will be inlined, that induction variables will be eliminated (or strength reduction will be performed), and that innermost loop parallel code will be vectorized (without manually writing SSE intrinsics). This can make the generated code a lot more readable, hiding strides and pointer manipulation. When in doubt, check the generated assembly code with `g++ -S`.
- Use Address Sanitizer to debug your memory accesses. Range checks on vector and matrix accesses are expensive, but without them hard-to-debug memory violations can occur. Address Sanitizer is a compiler flag that introduces range checks on all memory accesses, with minimal overhead. Use `-fsanitize=address` to enable.
- On a Linux system, to discover the details of the machine you're working in, use `cat`

`/proc/cpuinfo`. That will list the logical CPUs, their current and maximum speed, the L3 cache size, and the support for SSE and AVX vector instructions. To discover all the levels of caches supported, look for the files in `/sys/devices/system/cpu/cpu0/cache`. You will find the size, associativity and type (instruction, data or both) of the cache, as well as what CPUs share it.