CS 243 Homework 6 Solutions

Winter 2024

Due: March 6, 2024 at 11:59pm

Directions:

- This homework includes a Gradiance quiz, two written questions, and one programming assignment. The written part and the programming part should be submitted separately via Gradescope. No late days can be applied to the Gradiance quiz.
- You may use up to two of your remaining late days for this assignment.
- You can complete the programming part in pairs, but the written part should be done individually. If you look up any material in the textbook or online, you should cite it appropriately.

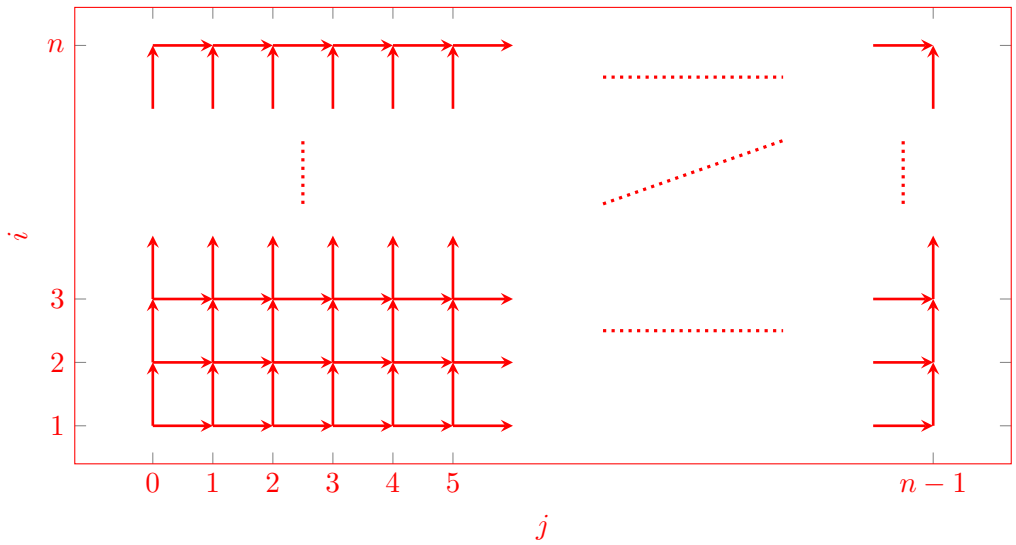
1 Written Questions: Pipelined Parallelism

Problem 1. Consider the following program.

```
for (int i = 1; i <= n; i++) {
    A[i, 0] = A[i-1, 0] * 2;
    for (int j = 1; j < n; j++) {
       A[i, j] = A[i-1, j] + A[i, j-1];
    }
}
```

1. Convert the program into a 2-deep fully permutable loop nest and write it down. Then draw the iteration space for the converted program. Include arrows to denote data dependencies that exist across iterations. (You don't need to draw out all the arrows, just enough to get a clear picture of the dependencies.)

```
for (int i = 1; i <= n; i++) {
    for (int j = 0; j < n; j++) {
        if (j == 0)
            A[i, 0] = A[i-1, 0] * 2;
        else
            A[i, j] = A[i-1, j] + A[i, j-1];
    }
}
```



2. Apply pipelined parallelism with blocking to your program. Assume your machine has 8 single-core processors, you choose blocking size $B = 16$ based on the cache properties of your machine. Also assume $n = 4096$. Show the generated code **for each processor** (use p as the processor ID, assume p ranges from 0 to 7) that is optimized for cache locality. You may use sync variables as presented in slide 8 of lecture 11. You may describe in text how sync variables are initialized without writing out the code. Your solution should have reasonable synchronization overhead.

```
// t[0]..t[7] are sync variables initialized to 0.
// begin of code executed on the processor of ID = p
if (p==0) p_synch = 7 else p_synch = p-1;
for (int ii = 1+16*p; ii <= 4096; ii += 128) { // parallelizable loop
    for (int jj = 0; jj < 4096; jj += 16) {
        if (ii == 1 || wait(t[p_synch] >= t[p]+16) ) { // +16 is necessary
            for (int i = ii; i < ii+16; i++) {
                for (int j = jj; j < jj+16; j++) {
                    if (j == 0)
                        A[i, 0] = A[i-1, 0] * 2;
                    else
                        A[i, j] = A[i-1, j] + A[i, j-1];
                }
             }
        }
        t[p]+=16;
    }
}
// end of code executed on the processor of ID = p
```

Since we only have 8 processors, parallelizing the outer loop is enough to maximize the parallelism. Parallelizing both loops will introduce more synchronization overhead.

Alternative Solution:

```
processor pid = {0, 1, ..., 7}
------------------------------
for(p = pid; p < 256; p += 8){
    for(int jj = 0; jj < 4096; jj += 16){

        // > instead of >= inside WAIT because jj starts from 0
        if(p == 0 || WAIT(t[p-1] > jj / 16)){
            for(int i = 16 * p + 1; i < 16 * p + 17; i++){
                for(int j = jj; j < jj + 16; j++){
                    if (j == 0) { A[i, 0] = A[i-1, 0] * 2; }
                    else { A[i, j] = A[i-1, j] + A[i, j-1]; }
                }
            }
        }
        t[p] += 1;
    }
}
```

Problem 2. Parallelization with Locality Optimization

Consider a multiprocessor with 8 processors, where each processor has two levels of caching, and there is a **significant overhead** in moving data between processors. It is therefore very important to maximize locality. The following function is a simplification of the ADI Integration Method; it is executed repeatedly in the full program. Your task is to parallelize the function, while maximizing locality. Assume that the array `A` is much larger than 8 times the size of the first-level cache in each processor.

```
 1 for i = 1 to N {
 2     for j = 1 to N {
 3         A[i,j] = 0.5 * (A[i,j] + A[i,j-1]);
 4     }
 5 }
 6 for i = 1 to N {
 7     for j = 1 to N {
 8         A[i,j] = 0.5 * (A[i,j] + A[i-1,j]);
 9     }
10 }
```

1. Without performing any code transformations, indicate which of the loops in this function are do-all loops? (Refer to the loops by their line numbers.) Suppose all such do-all loops are parallelized by inserting barriers at the end of each loop. Discuss the cache performance for the individual processors, as well as the synchronization and communication cost between processors.

   Loops at lines 1 and 7 are parallelizable. The first loop nest has good performance since individual processors have memory locality. The second loop nest has bad performance since every processor needs to work on the same block of memory (e.g., processor 1 writes to $A[1,1]$, processor 2 writes to $A[1,2]$), increasing communication costs dramatically.

   Additionally, there must be a barrier between the two loops, incurring synchronization cost. More importantly, the first loop partitions by rows, the second loop partitions by columns, requiring the whole array to be transposed, which is extremely costly.

2. Does this function have any communication-free parallelism (i.e., can you transform the program into a **single** nested loop nest whose one or more outer loops are do-all loops)? If so, show the SPMD code. If not, briefly explain why.

   Notation: use i, j to describe the first loop nest, and i', j' the second loop nest.

   There is no communication-free parallelism. Suppose for contradiction that there is, and we have processor mappings $p(i,j) = xi + yj + z$ for the first loop nest and $p'(i',j') = x'i' + y'j' + z'$ for the second loop nest. We know that iteration (i',j') depends on iteration (i,j) in the first loop nest and also $(i'-1,j')$, and iteration (i,j) depends on $(i,j-1)$. Communication-free parallelism implies that dependent iterations must be run on the same processor, so we have

$$p(i,j-1) = p(i,j) = p'(i',j') = p'(i'-1,j')$$
$$xi + y(j-1) + z = xi + yj + z = x'i' + y'j' + z' = x'(i'-1) + y'j' + z'.$$

The first equality implies $y = 0$. The second implies $x = x'$, $y = y'$, $z = z'$. The third implies $x' = 0$. In other words, the processor mappings $p(i, j) = p(i', j') = z$ map all iterations to a single processor, so there is no parallelism.

3. Does this function have pipelined parallelism? If so, show the transformed code with as many outermost fully permutable loops as possible. (You only need to show the transformed sequential code, there is no need to parallelize it with synchronization.)

   Yes.

   Note that the following loop does not give the right answer because there is an anti-dependence from the read of A[i,j-1] and the store of A[i,j] in the second loop.

```
for i = 1 to N {
    for j = 1 to N {
       A[i,j] = 0.5 * (A[i,j] + A[i,j-1]);
       A[i,j] = 0.5 * (A[i,j] + A[i-1,j]);
   }
}
```

   One solution is to remove the anti-dependence by saving the value that would otherwise be overwritten.

```
for i = 1 to N {
    tmp = A[i, 0];
    for j = 1 to N {
       A[i,j] = 0.5 * (A[i,j] + tmp);
       tmp = A[i,j];
       A[i,j] = 0.5 * (A[i,j] + A[i-1,j]);
   }
}
```

   Another solution that only uses affine partitioning is to shift the second loop nest by 1 to satisfy the anti-dependence.

```
for i = 1 to N+1 {
    for j = 1 to N {
       if (i <= N)
          A[i,j] = 0.5 * (A[i,j] + A[i, j-1]);
       if (i > 1)
          A[i-1,j] = 0.5 * (A[i-1,j] + A[i-2,j]);
   }
}
```

   We give full credits to students who recognize that it is a full permutable loop nest even if they do not handle the anti-dependence.

4. Describe how you would parallelize this function. Discuss the cache performance for the individual processors, as well as the synchronization and communication cost between processors.

   Parallelize with blocking. This hinders some parallelism, but in exchange it:

   (a) reduces communication significantly since now processors only need to communicate the "edges" of the blocks to each other;

   (b) provides good cache locality, as long as we make sure the same processor works on adjacent blocks;

   (c) minimizes synchronization since two processors only sync once every B elements processed, where B is the block size.

2 Programming: Optimizing Numerical Code for Parallelism and Locality

Directions: The goal of this problem is for you to learn how to write parallel code, and how to optimize parallelism in existing numerical code. Your task will be to take some existing, serial code, and write the corresponding parallel code to achieve the best performance.

Download the starter code from the course website, complete the two tasks, and submit the zip file containing your work to Gradescope.

Setting up the environment

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

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

Rice machines are similar to myth machines used in previous assignments. However, we use rice for this homework because myth is not appropriate for intensive CPU use. The home directory of myth machines can be accessed from rice through path `~/afs-home`.

Note: not all rice 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`).

Using your own computer. You must have a GCC-compatible C++ compiler (such as GCC itself or Clang), which must support C++11 and OpenMP 4.0. Any modern C++ compiler, including the one in Ubuntu 16.04 LTS (or later) will do. However, since grading will occur on rice, take care not to use language features that are too new or non-portable.

Parallel C++ code

This assignment uses **OpenMP** to allow C++ code to run in parallel. OpenMP consists of a set of "pragmas" or directives that allows us to write code like this:

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

What this statement means is that the entire loop will be 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. OpenMP allows us to not worry about threading or locking, and we just need to write the outermost loop parallel code, as discussed in class.

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

The clause `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 parallelize over more dimensions using the `collapse(·)` clause:

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

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.

For more resources on OpenMP, you may consult its specification[2] and syntax reference card[3], some example code[4], and a guide on loop scheduling[5].

Task 1: Basics of Writing Parallel Code with OpenMP

To start, we will consider Matrix Multiplication, a very widely used routine in many real-world applications 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 of arbitrary types (most commonly `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 naïve, parallel naïve, and parallel blocked. You can choose which one is invoked by modifying the starter code.

[2] https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf
[3] https://www.openmp.org/wp-content/uploads/OpenMP-4.0-C.pdf
[4] https://www.openmp.org/wp-content/uploads/openmp-examples-4.0.2.pdf
[5] http://www.inf.ufsc.br/~bosco.sobral/ensino/ine5645/OpenMP_Dynamic_Scheduling.pdf

1. Test matrix multiplication between two matrices both of size 1000×1000. What is the average time? Which is the fastest algorithm?

2. Test matrix multiplication between matrix A of size 1000×10, and matrix B of size 10×1000. What is the average time? Which is the fastest algorithm?

Task 2: 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 one cell (unit) of the long short-term memory (LSTM) recurrent neural network (RNN). 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 ($\sigma(x) = (1 + \exp(-x))^{-1}$), tanh is the hyperbolic tangent function, and $\circ$ denotes component-wise multiplication. W, U and B are the learned parameters of the LSTM. W and U are rectangular matrices, while B is a row vector.

Vectors 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)[6]. 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 gates f, i, o, and 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 naïve 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 to:

1. Write an efficient serial implementation of the LSTM in `lstm.cpp`. Efficient here means that the implementation is friendly to cache and memory. You could also explore techniques that may improve instruction-level parallelism and register allocation.

[6]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.

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 plot (average time vs. number of processors) for an input and output of size 5000, batch size of 100/200/500 (choose at least one batch size among the three options), 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_THREADS=`x `./lstm` ..., where x is the desired number of threads. Note: not setting `OMP_NUM_THREADS` explicitly does not mean there is only one thread.

   To draw the plot, you can use any plotting library (matplotlib, pgfplots, MATLAB, or R), or an application such as Google Sheet. Please also label your axes, especially if you use a non-standard scale.

In this task, you should try to write efficient code by applying 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 choices 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. 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/AVX 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 `lscpu`. That will list the logical CPUs, their current and maximum clock 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/cpu`X`/cache` (where X is the CPU index). You will find the size, associativity, and type (instruction, data or both) of the cache, as well as what CPUs share it.