ECE 451-566: Introduction to Parallel and Distributed Computing, Fall 2020

Instructor: Maria Striki

Homework 2: (10 + 16 + 8 + 9 + 14 + 24) = 81 points

Assigned: Sunday 11-1-20 Due: Thu 11-12-20, 8.00pm

Problem 1A: (10 points)

Assume some variant of multiplication operation involving two square matrices labeled via the symbol "@". Both *a* and *b* are of dimensions W x W. They can both fit in DRAM (main memory) but neither can fit entirely in the cache. The entry of one line of cache is 8 words or 64 bytes < W.

```
# Do here all proper initializations on matrix c.
for (i=0; i<dim; i++)
       for (j=0; j<dim; j++)
              for (k=0; k<dim; k++)
                     c[i][j] += a[i][k] @ b[k][j];
```

1.(4 points) How does the peak achievable performance of the instruction stream differ than the average achievable performance?

2. (3 points) What are the possible peak achievable performances attained in native C and what in ISPC, based on the way we store the matrices in memory? Please explain.

3. (3 points) Is there any known programming language or compiler/assembler that can increase the maximum achievable performance in any case/strategy? Which one and how? What assumptions do we need to make to ensure that we can achieve up to twice the peak achievable performance of native C?

Solution:

Problem 1B (10 points): level: moderate

i) **(4 points):** Please provide DETAILED justification about why the "Footprint" version (2nd version) of the Grid-Solver that trades off footprint for barriers is correct and how exactly do we achieve equivalent operation. Describe exactly why and when the array diff gets changed to what by who. Which threads are affected and how. What is the scope of the variables involved? Use the code quoted below to map your justification to specific lines of code.

ii) **(3 points):** Write a detailed proof of why it is sufficient to use 3 states in variable diff and why not fewer or more states to remove the dependences shown below.

iii) **(3 points):** And why do we still keep one barrier in this code? What should happen if we remove it?

```
int     n;               // grid size
bool    done = false;
LOCK    myLock;
BARRIER myBarrier;
float diff[3];  // global diff, but now 3 copies

float *A = allocate(n+2, n+2);


void solve(float* A) {
  float myDiff;   // thread local variable
  int index = 0;  // thread local variable

  diff[0] = 0.0f;
  barrier(myBarrier, NUM_PROCESSORS);  // one-time only: just for init

  while (!done) {
    myDiff = 0.0f;
    //
    // perform computation (accumulate locally into myDiff)
    //
    lock(myLock);
    diff[index] += myDiff;    // atomically update global diff
    unlock(myLock);
    diff[(index+1) % 3] = 0.0f;
    barrier(myBarrier, NUM_PROCESSORS);
    if (diff[index]/(n*n) < TOLERANCE)
      break;
    index = (index + 1) % 3;
  }
}
```

Idea:

Remove dependencies by using different `diff` variables in successive loop iterations

Trade off footprint for removing dependencies! (a common parallel programming technique)

Solution:

Problem 2: (16 points) Parallel assignment and Grid Solver

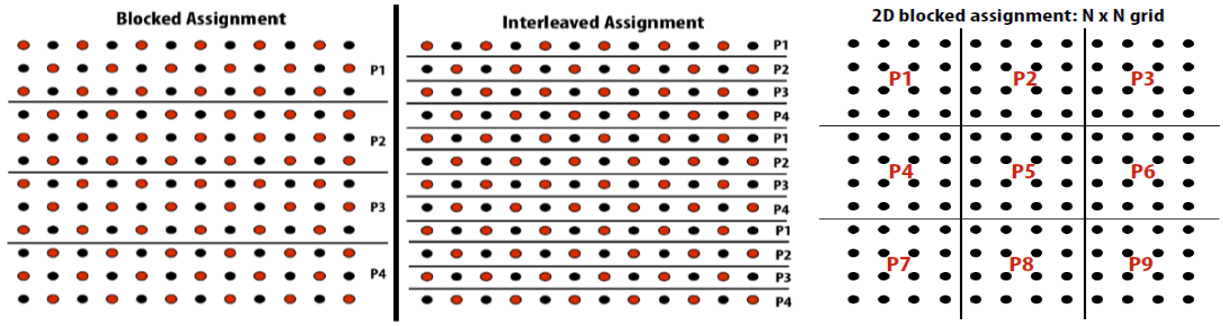

Fig.1: Interleaved vs. blocked assignments of work to multi-cores for the Gauss-Seidel Computation: N x N elements

Part 1 (5 pts) In the Red-Black manipulation, what is the **arithmetic intensity (AI) per processor** and **as a whole** in each **case**? Assume P processors in each case. The grid solver has the same size: N x N. In your illustrations, schemes 1 and 2 are assigned 4 processors (but we want to generalize for arbitrary P). Scheme 3 is assigned 9 processors in a blocked fashion (but we want to generalize for arbitrary P).

Your goal is to develop 3 closed analytical formulae of **AI** for each case, using parameters N, P. Please ignore marginal cases (edge of grid). Which scheme ends up in the highest/lowest **AI** and why?

NOTE: Arithmetic intensity (AI) is defined as the **ratio of computation** divided by **the communication cost**. **AI** is the number of elements computed, divided by the number of elements communicated. The computation of the per element avg. is done in 1 clock cycle, and any request for communication across processors has cost 1.

Part 2 (3 pts) Propose 2 strategies to solve the Gauss-Seidel problem via purely sequential programming. Are there 2 valid approaches? If so which is better and why? Is there any criterion as to why one is a better strategy over another? How do we decide beforehand? Can we?

Part 2B (3 pts) : Let's now answer the questions of Part 2 by assuming the following scenario: We can refer the every grid element (sensor) as an element [*i*, *j*] of a bi-dimensional array where *i*: row, *j*: column. Assume that the cache of each processor is very small and can hold up to 4 elements at a time, following a Row Major order. The sensors have to do the computation shown below, but instead of the Gauss Seidel problem they just compute the average result across the grid.

i) Assume that every sensor must compute the primality of the following integer: [*i**j]
ii) Assume that every sensor must compute the primality of the following integer: [*i*^4 + *j*^2]

Part 3 (5 pts) Back to the Gauss-Seidel problem again. How many elements can you process SIMULTANEOUSLY during each clock cycle under the strategy illustrated in each of the three cases? Consider the following 2 scenarios for the processor:

1) Each of the processors P is dual core, hyper-threaded, with 4-way SIMD.
2) Each of the processors is single core, single-threaded, with no SIMD capabilities

Solution:

Problem 3: (8 points)

You plan to port the following sequential C++ code to ISPC so you can leverage the performance benefits of modern parallel processors.

```
float input[LARGE_NUMBER];
float output[LARGE_NUMBER];
// initialize input and output here
for (int i=0; i<LARGE_NUMBER; i++) {
      int iters;
      if (i % 16 == 0) iters = 256;
            else iters = 8;
      for (int j=0; j<iters; j++)
      output[i] += input[i];
}
```

We inspect the following CPUs all for the same price:
_ 4 GHz single core CPU capable of performing one FP addition per clock (no parallelism)
_ 1 GHz quad-core CPU capable of performing one 4-wide SIMD FP addition per clock
_ 1 Ghz dual-core CPU capable of performing one 16-wide SIMD FP addition per clock.

If your only use of the CPU will be to run your ISPC port of the above code as fast as possible, which machine will provide the best performance for your money? Which machine will provide the least? When considering execution time, you may assume that (1) the only operations you need to account for are the floating-point (FP) additions in the innermost loop. (2) the ISPC gang size will be set to the SIMD width of the CPU. Consider the execution time of groups of 16 elements of the input and output arrays.

Solution:

Problem 4 (9 points):

Part 1: (3 pts)
What are the two main differences between pthread_join and pthread_exit?

In the Pthreads library, the prototype of `pthread_join` is:

```
int pthread_join(pthread_t tid, void **ret);
```

and the prototype of `pthread_exit` is:

```
void pthread_exit(void *ret);
```

Part 2: (6 points)

Please go over the Pthreads demo where we compare assigning tasks to as many threads are tasks versus a fixed limited number of threads (persistent threads) [discussion in class and through pre-recorded session].

(a) Study the role of variable: `num_tasks_performed_by_worker_thread` for all 4 threads. The demo compares the sum of these values vs the total number of tasks: `thread_data_num_tasks`, both when num_tasks = 10,000 and 100,000. We noted the same discrepancy: the tasks calculated by all threads together amounted to 3 more than the cap (`thread_data_num_tasks`). Was this random? Why is this happening?

(b) Study the role of variable: `thread_data.next_task`. Does it play a role in the situation observed in the previous question? What sort of variable is this? Global, shared, private? Does it have any synchronization primitive associated with it?
Describe what happens if we assume it does - provide a scenario (if there is not such in the demo).
Describe what happens if we assume it does not - provide a scenario (if there is not already such in your demo, otherwise explain how this is handled in the demo).

Problem 5 (14 points):

The below instruction stream illustrates a sequential computation and you are told that this is part of a convergence problem (such as the Gauss-Seidel) and must be parallelized in the best possible way.

```
# Do here all proper initializations on matrix c.
for (i=0; i<N1; i++)
       for (j=0; j<N2; j++)
                    a[i][j] = f(a[i-1][j-1], a[i-1][j], a[i][j-1]);
```

Q1 (5 points) Identify the dependencies and identify which loop iterations can be parallelized or how they can be modified to be parallelized. Propose at least 3 different solutions to this problem and modify the pseudo-code above (loops) accordingly. Discuss the nature and strategy of your solution with adequate details.

Q2 (3 points) What is the maximum speedup we can achieve, taking into consideration the three solutions computed above.

Q3 (3 points) Assume that we have 4 underlying cores to implement this problem. Provide high-level pseudo-code or description for parallel implementation in shared address space model.

Q3 (3 points) Assume that we have 4 underlying cores to implement this problem. Provide high-level pseudo-code or description for parallel implementation in message passing model. You figure out the preferred distribution of data to your cores and pinpoint the points and the mode of communication.

Solution:

From the following 2 problems select only 6A OR 6B for your group:

Problem 6A (24 points): programming exercise (it will be developed later to a fully-fledged project question).

Simple Primality Test: The simplest primality test is *trial division*: given an input number, *n*, check whether it is evenly divisible by any prime number between 2 and $\sqrt{n}$ (i.e. that the division leaves no remainder). If so, then *n* is composite. Otherwise, it is prime.

What to do: Write a simple multi-threaded program (using C and Pthreads library) that outputs prime numbers. The prime numbers will be printed to a shared output array/list/file. You can even produce a primality array, that for the same index corresponding to an array element outputs (1=primary, 0=composite). The program should work as follows: as input, provide an array of 100 ODD integers, varying from 400-800, generated randomly. The program will check each of those integers for primality, output the result, and measure how long it takes to completion, in one of the following ways:

1. Generate 100 Pthreads, each of which will check each of the array integers for primality.
2. Generate 4 Pthreads, each of which undertakes a block of the array elements, and
3. Generate 4 Pthreads, each of which undertakes one available element of the array every time, and once computation is complete, they undertake the next available element, and then the next (THIS STEP IS OPTIONAL TO IMPLEMENT – EXTRA CREDIT STEP).

Also, check which subset of the above primes have the following property: the number that is derived by reversing the digits is also prime (e.g., 79 and 97). Provide these primes in separate output.

Questions and Results:

Q1. Execute your program using any of the three implementations (again, third is extra credit). Compare the execution times. Which implementation is faster? How do you explain it?

Q2. If you implemented the 3 option, discuss your findings, compare option 3 with option 1 and option 2. (OPTIONAL STEP – EXTRA CREDIT).

Q3. Even if you did not implement step 3 and hence did not answer Q2, please attempt to provide insight as to whether you expect strategy 3 to be better than strategy 1 or 2 or not. If not, what are the circumstances under which you are convinced that strategy 3 can be better than strategy 1 and 2.

Note: In your submission please submit your code quoted in your HW solutions with a ReadMe file with instructions on how to run your code. For this code to work you will need to watch the pre-recorded session on Pthreads and/or read the related lecture slides and check out the demo discussed in class. You are required to use minimal number of Posix Pthread commands: create a number of threads and exit them, and you may need to use lock mechanisms, such as the Pthread_mutex_lock and Pthread_mutex_unlock.

Solution:

Problem 6B (24 points):

The below instruction stream illustrates a sequential computation that solves a linear problem.

```
# Do here all proper initializations.
for (k = 0; k < N; k++)
       for (i = k+1; i < N; i++) {
              l = a[i][k] / a[k][k];
              for (j = k+1; j < N; j++)
                     a[i][j] = a[i][j] – l*a[k][j];

       }
```

Qp1 (5 points): Design and sketch a parallel program to execute the above computational kernel to a multi-core platform that supports the shared address scheme. You may provide two different solutions, one focusing on the task parallelism and another focusing on the data parallelism logic.

Qp2 (5 points): Now follow the task parallel logic and sketch the dependencies. Can you see what strategies help you achieve the maximum speedup?

Qp3 (5 points) : Identify in the graph where proper synchronization mechanisms are required to ensure correct execution and describe which those mechanisms can be: locks, barriers, your own customized schemes? Why?

Qp4 (4 points): If we are given M processors/cores/threads to distribute the computations of the elements of this grid, how will you select to implement your tasks? Using a blocked or interleaved assignment? Why?

Qp5 (5 points): Going further into the implementation of the above kernel, identify which loops can be parallelized and write the pseudo-code that parallelizes them. You may use ISPC-like model for your pseudocode. You may also provide more than one solutions if you can think of more.

Solution: