Homework 2

Brian J Gravelle

May 31, 2017

1 Dependencies

1.1 Question 1

What is sequential consistency and why is it important?

Sequential consistency is the idea that a parallel program should produce a result consistent with that produced by the sequential version. Ensuring sequential consistency is important to guarantee a correct results, help identify dependencies, and prevent undesired non-determinism.

1.2 Question 2

What is a dependency? Do dependencies only arise in shared memory parallel programming?

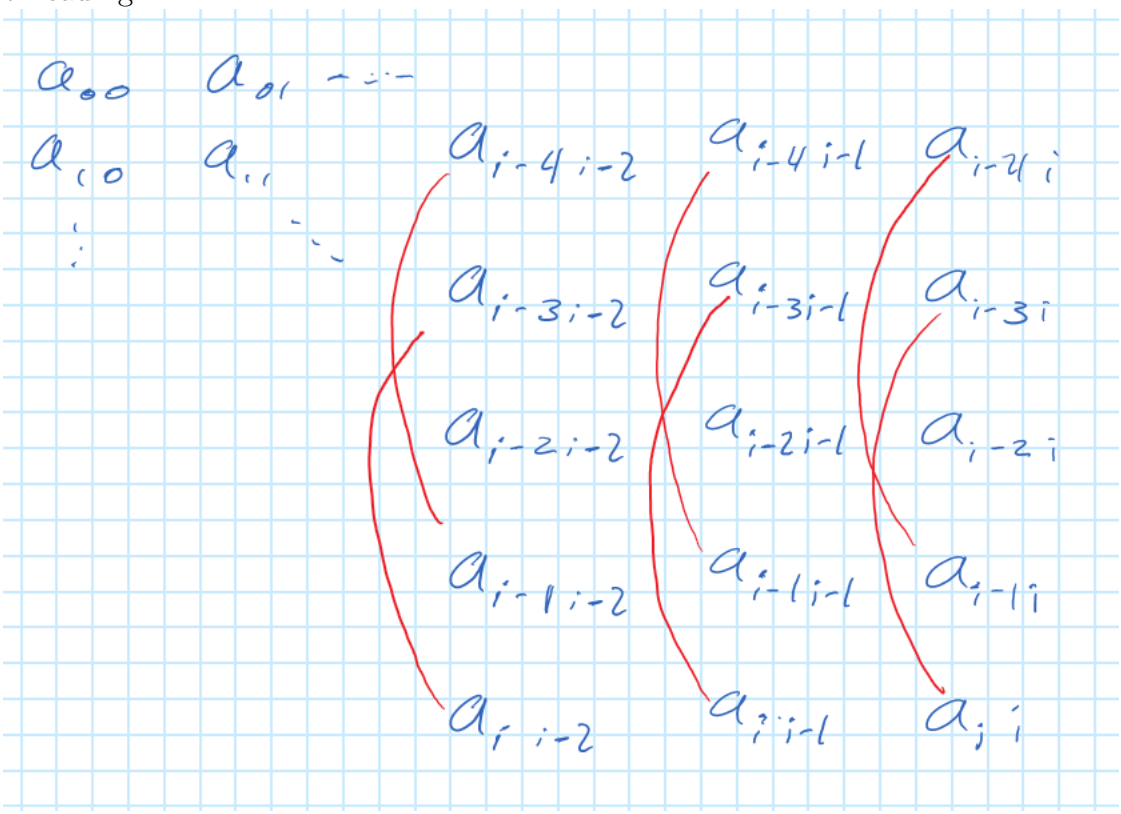
Dependencies are logical connections between parts of a computation where correct computation of one section relies on prior completion of the other section. Dependencies are very noticeable in shared memory programming because the threads are editing the same memory, but as a logical construct they are part of all parallel and sequential programs.

1.3 Question 3

Consider the following nested loop:

```
for i = 4 to 100 do
    for j = i-2 to i do
      a[i][j] = a[i - 3][j];
    end
end
```

Suppose the loop is executed on a shared-memory machine using multi-threading.



There is a flow dependence between the third item within each of the columns but no dependencies between the columns. so there is as much parallelism as there are columns. Within the columns there is enough parallelism for 3 tasks.

```
for i = 6; i < 100; i+=3 do
    #pragma omp parallel for
    for k = i-2 to i do
        for j = k-2 to k do
```

```
            a[k][j] = a[k - 3][j];
        end
end
```

2 Patterns

2.1 Question 1

One algorithm used in image blurring is to loop through each pixel in an image, gather the values of the current pixel's neighbors and average them into the current pixel. If we wanted to do this in parallel, what pattern might we choose (briefly explain why you suggest a particular choice).

I am inclined to use the stencil pattern since the results in one element are based on the corresponding input element and the inputs around it. While a fork-join pattern may be part of the implementation of the stencil, that pattern tends to reflect task-parallelism rather than the data-parallelism addressed by this question. map is similar to stencil but refers to situations in which each output element only depends on the corresponding input element. Reduce would also be undesirable as it produces one value based on all elements in the input set. Although each stencil section could be treated as a reduce.

2.2 Question 2

Given an array of data, what parallel pattern might one use to copy a subset of that data with another array of indices where the indices correspond to our input array (briefly explain why you suggest a particular choice)?

I would use pack because the pack pattern is designed to take an array of data and array of indicies and make a new array with just the data corresponding to the indicies.

3 Map Pattern

3.1 Question 1

What makes map such a desirable pattern in parallel computing?

The map pattern is embarrassingly parallel in the sense that there are almost no dependencies which means that as data and processing units are added the program can obtain nearly linear speedup.

3.2 Question 2

Why might one want to "fuse" a series of maps on the same data, as opposed to performing multiple maps one after the other?

Although the map pattern is embarrassingly parallel there is still some overhead incurred in distributing data, managing "iterations", and synchronizing the processing units at the end. If multiple maps are fused this overhead doesn't need to be repeated.

4 Collective Pattern

4.1 Question 1

Given a list of words, write a (pseudo-code) function to sort them in parallel in dictionary order.

```
list sort_words_in_parallel(words, num_words)

        if(num_words > 1)
                temp_num_words = num_words
                while(temp_num_words > 1)
                        temp_num_words = temp_num_words/2
                        fork(half of words,temp_num_words)
                        words = other half
                while(children)
                        half_list = join most recent child
```

```
                    words =merge(half_list, words)

    return words
```

4.2 Question 2

Why would we want to "tile" our data when performing a reduce or a scan? Doesnt this lead to less parallelism than non-tiled data?

For very large problems a single row of a matrix may not fit nicely into the cache, so if computation is performed of an entire row at a time it may result in frequent cache misses. Tiling reduces the parallelism and complicates code, but limits each chunk of computation to a more reasonable size of data.

4.3 Question 3

Only associativity of a combiner function is required to parallelize reduce and scan. Why is this the case?

If the combiner function is associative then the order that data is added doesn't matter so race conditions aren't an issue. if the combiner is not associative then the computation would need to be sequential to ensure consistency.

5 Data Reorganization Pattern

5.1 Question1

In general, what is the purpose of data reorganization and of having a parallel pattern to do it?

Sometimes data is organized in a way that in not ideal for a particular computation. For example, a matrix may be defined row-wise when the computation is column wise, or an array may contain lots of extra data when only some elements are needed. In these situations reorganizing the data before the computation begins can be a good way to improve cache

performance, reduce memory usage, or simplify the computation. Ideally the reorganization will also happen in parallel to minimize the overhead.

5.2 Question 2

In many physical system simulations, multiple data values might be stored together in a data structure for each cell in the system, representing certain physical properties. For instance, suppose you are simulating the atmosphere dury 24 hurs for a 10km x 10km x 10km volume where each cell is 10m x 10m x 10m in size (1B cells total) and contains data about velocity, temperature, and ozone content. The data structure might look something like:

```
struct Cell {
  float velocity , temp, ozone;
};
Cell atmosphere [1000 ,1000 ,1000];
```

Calculating the next values for each Cell can be done simultaneously for each field. That is, 1/3 of the threads could be used to update the velocity, 1/3 to update the temperature, and the remaining 1/3 to update the ozone content. (Note, each update probably will involve a stencil computation, but that is not the point of the problem.) Suppose we want to then execute all of the updates in parallel.

What effects could there be on performance given this memory access pattern?

This memory pattern organizes data so that the values for each cell are located consecutively in memory. As a result if a thread reads one piece of data about a cell the other two fields will most likely be read into cache as well. Since each thread is limited to computation of velocity, temp, or ozone this pattern will result in a great deal of wasted cache space and 2/3s of every cache line will go unused.

How would you resolve memory performance issues using a parallel data reorganization pattern?

Rather organize the data as an array of structs, the programmers should consider organizing the data as a struct of arrays. This design would ensure that the data needed by each thread would be located consecutively in memory resulting in more efficient cache usage.

6 Stencil Pattern

6.1 Question 1

Imagine working on data that can be represented in 3 dimensions as 1000 x 1000 x 1000 cells. We would like to perform a stencil operation on each cell. The stencil is a 3 x 3 x 3 cube around each cell (including the current cell) for a total of 18 cells per stencil operation (19 including the center point, see example). When partitioning this data among 8 threads, each thread will be responsible for a 500 x 500 x 500 chunk of data.

Describe what the ghost cell regions will be like in this computation.
for this computation each thread will be responsible for one "corner" of the data cube meaning that the ghost cells will be those in the center on the boundary between the threads. Each thread cube has three sides on the boundary and three sides with ghost cells. The ghost cells consistent of three 500x500 cross sections, three 500x1 corners, and 1 point at the junction of all three boundary sides. There are 751501 ghost cells in total for each thread.

Suppose we would like to use more threads in this computation. How many threads would we need if we wanted each "chunk" to be 250 x 250 x 250? 125 x 125 x 125? (Show work)

$250 * 250 * 250 * nthreads = 1000 * 1000 * 1000$
$nthreads = \frac{1000000000}{15625000}$
$nthreads = 64$

$125 * 125 * 125 * nthreads = 1000 * 1000 * 1000$
$nthreads = \frac{1000000000}{1953125}$
$nthreads = 512$

Using your answers from the previous questions, describe how the halo regions change relative to the rest of the data. Consider the ratio of ghost

cells to non-ghost cells. How does this change effect our computation?

The amount of non-ghost grows more rapidly than the ghost as the block size grows. Since the ghost regions corresponds to communication the ratio of ghost to non ghost data can be used as a proxy for the communication to computation ratio. Once this ration is low enough then the communication latency can be entirely hidden by the computation; however, going lower than that could imply that the program doesn't take full advantage of the available parallelism.

6.2 Question 2

Given a halo region in a stencil computation, why might we want to increase the "depth" of our halo region? Why might making this "depth" too large be a bad thing?

We can increase the depth of the halo region by redundantly computing some of the layers around the chunks. This technique trades communication overhead for computation overhead by reducing the frequency that the halo region must be communicated. As a result the halo region will grow slightly with each layer that is added and each thread will be required to perform more computation that is redundant with the computation that others are doing. These overheads will at some point overtake the gains and cause the computation to slow. Additionally, the power efficiency of the application will be impacted differently than speed efficiency so there may be competing goals.

6.3 Question 3

Consider the following nested for loops:

```
for (i=1; i < n; i++) {
    for (j=1; j < m; j++) {
       A[i][j] = f( A[i-1][j], A[i][j-1], A[i-1][j-1], B[i][j] );
    }
}
```

Despite the dependencies between loop iterations, this can still be parallelized. Briefly describe how this operation would be parallelized.
(Tip: this is known as a recurrence pattern and is covered in the book.)

You have to find the plane that cuts through multiple items in the grid such that all dependencies of those items fall to one side of the grid. All the item that said plane cuts through can be parallelized together. It is called a separating hyperplane and is rather nifty.

7 Fork-Join Pattern

7.1 Question 1

Write a Fork-Join parallel implementation (pseudocode is acceptable) to compute f(100) for the following recursive sequence:

f(x)=2*f(x 1)+3 where f(4)=2

I think this could be parallelized by tweaking the recurrence relation and doing a pipeline pattern, which could be implemented with fork-join syntax. However it really doesn't fit nicely into the fork-join pattern because of there is only one recurrence so if you fork that then the parent would be idle. Honestly this would be better written as a loop which would make the dependency that impedes parallelism pmore obvious.

A more suitable function for fork-join

f(x)=2*f(x 1)+3*g(x−2) where f(4)=2 and g(6)=1

7.2 Question 2

Is parallel slack necessary in fork-join parallel solutions? Briefly explain your answer.

Parallel slack is not necessary, but it can improve the efficiency. Because the fork join pattern adds new processing units then joins them back together, the amount of available parallelism changes significantly over the course of the program. As a result, if no parallel slack is available, then the resources

will only be fully utilized during a small portion of the computation so a lot of resources will be idle.

8 Pipeline Pattern

8.1 Question 1

What happens to available parallelism when the number of data items in a pipeline increases, but the number of serial tasks in a data item remains constant?

In this case parallelism remains constant.

8.2 Question 2

What is the main factor that limits a pipeline's parallel scalability?

A pipeline's scalability is primarily limited by the number of stages that it can be broken into.