MVP Assignment 5 - Threads

No due date - optional

Group room: FIXME
Group number: FIXME

1 Matrix Inversion - Preliminaries

You will find an updated `pmatrix.c` file that contains two matrix inversion algorithms. We will improve a bit our previous parallelization of the matrix inversion using a finer data dependency analysis. The first matrix inversion is the one you had previously and the second is using copy as I mentioned in one lecture. It turns out that if you look more carefully at the data dependencies between the different loops (both main loops on k), we can reduce the synchronization. For the first k loop, finding the max pivot depends only on the current k row. Once that max is found, swapping the rows depends only on both rows involved. The division step depends on the current k row. As for the second k loop, every iteration depends on the last k row computed at the previous iteration – except the first iteration where the row is already ready.

Exercise 1: Copy your previous parallel inversion implementation with the barrier from assignment 3.

Exercise 2: Copy your previous barrier implementation from assignment 3.

Exercise 3: Understand the finer dependency analysis.

Exercise 4: *Rearrange the elimination step so that the threads compute the first $k+1$ column (for $A(i,j)$ only) and then compute the other columns.*

Exercise 5: One key for the improvements is to use the owner-compute rule.
Q: *Recall what the owner-compute rule is.*

A: ..

Exercise 6: *Rearrange your code to apply the owner-compute rule*, making sure that a given row is always taken care of by the same thread. Use modulo for that. It may be the case that you are already doing this. Make sure this is used even for pivoting.

Exercise 7: We will experiment with split-barriers later. *Implement the split-barrier in the place holder.* You will find it in the book.

2 Matrix Inversion - Improved First Loop

The idea is to have a condition per thread that says what the thread has done so far. The condition say if the thread has not done anything (its `tok[id] == 0`), if the first column of the current iteration has been computed (its `tok[id] == 1`), or if the whole row has been computed (its `tok[id] == 2`). In principle we want this for all rows but we do this for all threads and each thread takes care of several rows. The key is that if we need to access some row `i` we check the condition of `i % nb_threads` because we applied the owner-compute rule and we know that that thread is responsible for that row.

Exercise 8: For finding the maximum value of the current column we need to wait for the condition $tok[id] \geq 1$ for all threads. Then we know all first elements of all the rows are ready. For swapping the rows, we need to wait for the condition $tok[id] \geq 2$ only for the right rows. Since we use the owner-compute rule, the thread executing this code already owns one row, we need to wait for only one other thread.

Q: *Implement the needed condition variables.* Please note that you will have to reset these conditions before reusing them between each iteration. The right place for doing this is in the barrier when you reset the counter. That way, you avoid races.

Q: *Use them in the loop.* You will need to wait for both conditions and to set them to 1 or 2 after the (rearranged) loops in the elimination step. You can move your second barrier outside the loop.

3 Matrix Inversion - Improved Second Loop

Exercise 9: We will use the concept of interleaving computation and communication. First, rearrange the loop of the back-substitution step to have the first i iteration separated. Be careful to respect the owner-compute rule.

Exercise 10: This first iteration computes the row that will be needed for the next k iteration but not for the rest of the current i iterations! We can use a split-barrier here as seen in the course to take advantage of this.

Q: *Use your split-barrier here.*

A: Give me your code for the barriers and the parallelization of the two loope.

Exercise 11: The split-barrier is good but is still coarse. The right condition is in fact "is the k row ready"? We note that the last row is ready when we start and then the first "ready row" will be decremented (the loop on k goes from $n-1$ to 1).

Q: (Optional) *Implement this condition variable.* In the code you will find the place holder for the "kloop" condition. This is it. You can set the current ready index k_{ready} and wait for the condition $k \geq k_{ready}$.

Q: (Optional) *Use that new condition variable instead of the split-barrier.* The thread that computes the $k-1$ row should set the condition and all threads should wait for the $k-1$ row to be ready before looping again.

```
1 /* Your relevant code goes here. NOT the whole file. */
```

Listing 1: Barriers and parallelization of the matrix inversion.

```
1 /* Your relevant code goes here. NOT the whole file. */
```

Listing 2: Added condition variable and the second loop using it (only).

Exercise 12: For this loop, it turns out that you can try a different data partitioning and eliminate all synchronization. This is done by partitioning on the columns instead of the rows. As you see in the back-substitution, if a thread owns a column, it always needs only that column to update its elements. However, this partitioning suffers from false sharing and accesses memory by strides.

Q: (Optional) *You can use a partitioning by block on the columns to avoid synchronization.* For this, the code inside the `#if 0 .. #endif` block can be useful.

4 Authors

I/We have solved these exercises independently, and each of us has actively participated in the development of all of the exercise solutions.

Name 1	Name 2
.....	.....
Signature	Signature
Name 3	Name 4
.....	.....
Signature	Signature
Name 5	Name 6
.....	.....
Signature	Signature
Name 7	Name 8
.....	.....
Signature	Signature