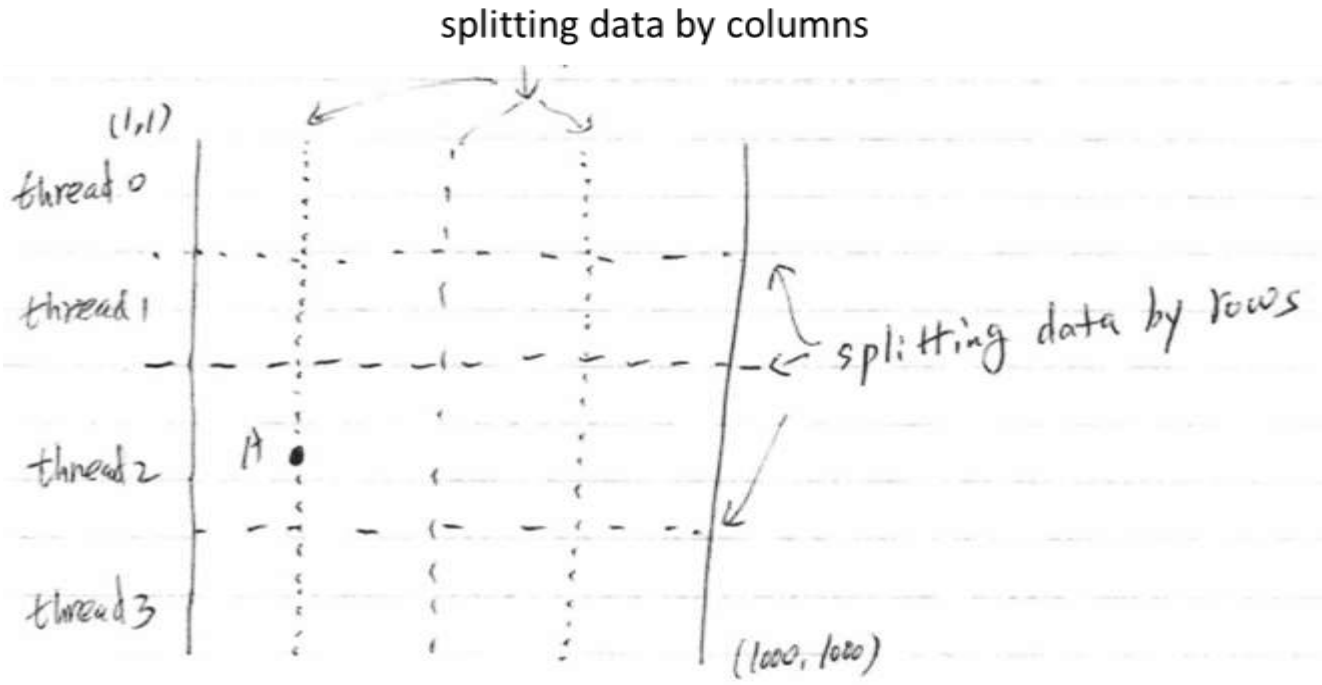
Homework 3 Sample Solution Explanation

(Not Optimized & Not 100% Correct Solution)

(Go over ~ in last part of Class 7 or Class 8 or Class 9, about 30 minutes)

Homework 3 sample solution is posted on course Canvas Homework folder/Homework3 Sample Solution (Not Optimized and not 100% Correct Solution) folder. Please note this sample solution is not only not optimized but also not 100% correct.



In figure above, we have a grid.

If we have 1 processor, the 1 processor will get all the data;

If we have 2 processors, we can split the data in the middle & each processor has half of the data.

If we have 4 processors, we split the data further in each half, and have each processor operates part of the data (1/4 of the data).

Each processor will get assign part of the data, depending on the thread ID and how it relates to all the other threads in the system. For each thread, we need to identify the 1st & the last index that the array has to go through.

For this problem, computations are done for each time step, depending on the temperature we had in the previous time step. For things here in the middle of the array (point to the middle of the array in the figure above), where data and all the neighbors for that data is just owned by one thread, then we don't have to wait for any other thread.

The problem is that there are these points on the boundary. If we are computing temperature at point A in the figure above, which is owned by a processor on the left (or this thread), then you have to wait for the data from the previous time interval that is owned by the neighboring processor (thread) on the right. If that's the case, then we have to wait, after each time step for the neighboring processor to compute all its temperature values, and then in the next time step, we have to read it.

The way to do this is by having two separate arrays, one for the current time step, and one for the previous time step. We compute values in the current time step based on values in the previous time step. Once we are done with each time step, then we'll switch the two arrays and make sure we wait for everybody to finish the previous time step, and then go to the next time step.

Let's see how this sample solution code works. It is not an optimized code. But it's a good start to at least see and easy to understand. Then we'll talk about some of the optimizations we could do to make it better.

(Show sample code)

These are the global parameters we are using.

(code line 11 – 20):

/* Global Defines */

4000 time steps

Maximum 16 processors

Range of temperature CMin 200, CMax 800 (CMax was initially assigned to a large value)

Cx & Cy are the coefficients

Xmax, Ymax: are the maximum X and Y coordinates.

We have 2 arrays, float type should work in this context, but double type is more accurate.

(code line 24, 25):

float old[Xmax][Ymax];

float new[Xmax][Ymax];

(code line 26):

int NumThreads; //Number of threads

(code line 29, 30):

struct timespec StartTime; // Then we have Start Time

struct timespec EndTime; // and End Time

(code line 39):

void Barrier()

{

…

…

…

}

// The Barrier() is the Barrier implementation. We are not going into that, that's the same thing we had in sum.c

(code line 60):

void* Temp(void* tmp)

/* Then this Temp is the function we will use in each thread. This is the function that computes the temperature that are split by rows. For each thread, we need to compute the starting address (starting row), and ending row. The data can be split by rows or by columns.

Q&A: Which one of splitting do you think will perform better? By row or by column?

There is a difference between split data by rows or by columns. Because usually the way the data is laid out in memory is typically by row major order, this is true for all programming languages except Fortran.

For row major order, location 0 or the 1st item in that data structure is data point (0,0), then (0,1) is the next one, then (0,2) is the next one.

So (0.0), (0.1), (0.2), …, (0.n)

After we are done with all 0's (the 1st row or row 0), then we start row 1 (the 2nd row).

If you split the data in rows, all the data will be contiguous in memory. If you split the data in columns, then you are breaking this continuous address space into smaller chunks. So you might not get the benefit of spatial locality. Spatial locality is good for things like prefetching.

So it's better to split the data by rows, unless you are doing the program in Fortran. Fortran does column major order, so in Fortran, it's better to split the data in columns.

(code line 137-140):

/* Step Calculation */

Count = (Xmax-1) / NumThreads;

Remainder = (Xmax-1) % NumThreads;

(code line 137-140) The last thread will get all the remainder rows as well in the Remainder calculation code line above.

(code line 78):

/* Job Execution for the number of steps as grid/number of threads*/

What each thread does is" it computes things for 4000 time steps.

This top "for loop" is for all time steps (code line 78), one time step at a time.

And with each time step, we are doing some operation for all the elements in our share of the array, to compute the temperature for those elements for the current time step.

(code line 79) j is the row index; (code line 80) k is the column index. j goes from start to end; k goes from 1 to Ymax (1000).

For each j & k, we compute the temperature at j & k, which is the old temperature plus coefficient of X (Cx) multiplied by the old left neighbor and the old right neighbor minus 2 times the current node and so on.

We do all these computations for the current node, then we assign the new array to the old array.

What is the problem if we do the following code line (code line 82)?

```
old[j][k] = new[j][k];
```

new [2,4] needs old [2,3], if we just overwrite old [2,3] with the new[2,3], then we don't have the old [2,3] to compute the new [2,4]. This is going to mess things up. This is not what we want.

What we wanted is to re-assign the value of old [2,3] after we are done with this iteration.

After this time step is done, then we can assign new to old. So we need to do it in a different nested loop and put "old[j][k] = new[j][k]" after the Barrier() which is after code line 86.

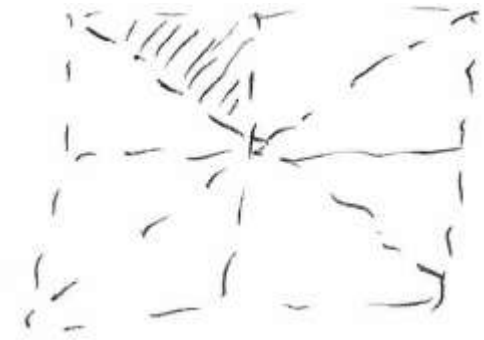
What are the other issues of this code?

This code is doing lots of additional computations which some computation could be saved or eliminated.

We are doing the same thing for each point. One way to optimize: (see figure below) we could potentially only do our computations on a quarter of the array; we can do computation on obviously half of the array, because the top half will have the same temperatures as the bottom half (mirrors).

We could also do it on a quarter of the array, because within one half, this quarter on the left mirrors the temperature on the outer quarter (the right quarter).

And if you want to get really creative, you could do it on a 1/8 of the array but the logic won't be as straightforward.



The other thing is we have 2 copies of the temperature array. Every time we are copying from one array to the other.

What needs to happen in the code is:

After the "Barrier();", we need to copy everything in the new array back to the old array.

It would be much simpler to avoid this additional copying by basically flipping the 2 pointers. We could have 2 arrays, and we have 2 pointers pointing to these 2 arrays, one of them points

to the old array, the other pointer points to the new array. At the end of each step, we can just flip the pointers and avoid all these copying stuff.

You just need to get the code working correctly, you are not going to be graded on these optimizations in the code.

(But for project, you do need to think about speed when you do your project. You need to choose the best algorithm to start with. One of the things the project will focus on is you need to come up with a good baseline serial or sequential algorithm and then after that, you need to come up with a parallel algorithm to improve upon. On the project, you need to start early.)