**PC40 hands on report**

**1D laplace solver**

**Exercise 1: print “hello world” in upc**

For the first exercise, we just had to print hello world with the upc language. The code was already written in the ex\_1.upc file.

Une image contenant texte

Description générée automatiquement

**Laplace 1D solver: Exercise 2:**

Here, we had to compile and execute a .c file.

Une image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquement

To do a test, I also launched the .c file with the classic commands to execute a .upc file, but on one thread. We can clearly see that executing a .c file like this is much slower for some reason (Maybe because the thread has to be initialized).

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

**Laplace 1D solver: Exercise 3 :**

Une image contenant texte

Description générée automatiquementThis is the first exercises where we have to code. First of all, we had to declare the arrays in the shared space.

We also had to change the for and the if statement to make it works with the threads.

Une image contenant texte

Description générée automatiquement

We can now execute it :

Une image contenant texte

Description générée automatiquementUne image contenant texte, équipement électronique, clavier

Description générée automatiquement

But as we can see, this is much slower than the .c version of this program. This happened because we are using a for loop with each one of the threads, and we also added an if statement in this loop. On the upcoming exercises, we will try to optimize the code.

Sometimes, the thread 0 prints the result before the other threads could finish. We can add a upc\_barrier right before thread 0 starts printing in order to wait for the other ones to finish.

**1D laplace solver : Exercise 4 :**

As said before, the if statement in the for loop in slowing down the execution. So we built a better for loop to get rid of the if statement.



We initialize j with MYTHREAD so it can start on the first element it has to work with. We also now have a for loop without the if in it, which should optimize the program.

Une image contenant texte

Description générée automatiquement

We can see that the modifications were useful, the program runs faster just by adding this small change.

**1D laplace solver : exercise 5 :**

In this exercise, we will use the upc\_forall loop to make the program faster. In fact, we will use the affinity of each thread, which should increase the time performance. We also had to work with blocksizes.

Une image contenant texte

Description générée automatiquement

We don’t have to initialize j on MYTHREAD like said before, because we are here working with affinities, so it would be pointless to do so.

Une image contenant texte

Description générée automatiquement

After compiling and launching :

Une image contenant texte

Description générée automatiquement

As we can see, the upc\_forall loop with the affinity was a uselful tool to get our program to run faster.

**1D laplace solver : exercise 6 :**

To make sure that all the threads finished its array swapping and iteration, we have to add some upc\_barrier in it. We simply add them at the end of for loop to make sure that they all finished before moving on.

Une image contenant texte

Description générée automatiquement

However, this is now much slower because of the 10000 iteration we are going through each time we enter in the for loop. The 2 upc\_barrier could also be slowing down the process.

We don’t need another upc\_barrier statement to make sure that thread 0 will start printing before the other ones, because there’s already a upc\_barrier at the end of the loop.

Une image contenant texte

Description générée automatiquement

This time, the program runs a bit slower than before. This is due to the 10000 iterations as said before.

**1D laplace solver : exercise 7 :**

To solve this exercise, we have to use the function in the upc\_collective.h library : upc\_all\_reduceD( &diffmax, diff, UPC\_MAX, THREADS, 1, NULL, UPC\_IN\_ALLSYNC | UPC\_OUT\_ALLSYNC );

With this function, we’ll be able to get a value diffmax, which is the max of all the diff values in the local memory of each thread.

We can then run the program:

Une image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquement

We can see that the diffmax value converged to 0.000048 at the end of all the iterations.

However, it is now running at a slower rate.

**2D\_heat conduction**

**2D\_heat conduction: C program**

To start off, we’ll be compiling and running the c version of the program :

Une image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquement

Une image contenant texte, cité

Description générée automatiquement

As we can see, for N=30 we have a 8,46 µs/iteration, for N=62 we have 32,2 µs/iteration and for N=94 we have 81,2µS/iteration.

In the following exercises, we will try to optimize the program in upc.

**2D\_heat conduction : first upc implementation**

Une image contenant texte

Description générée automatiquementIn this exercise, we had to convert the c code into a upc code. To do so, we first had to define a blocksize and define the arrays as shared.

Une image contenant texte

Description générée automatiquement

We also had to run the iteration a single time. That’s why we are launching the initialisation only if we are using the thread 0.

Une image contenant texte

Description générée automatiquementWe changed the for statement in the c file to a upc\_forall loop in order to work with the affinity of the thread for a better optimisation.

I also added the bupc\_allv\_reduce\_all function in the bupc\_collectivev.h library to get the best value of dTmax out of all the threads after each iteration.

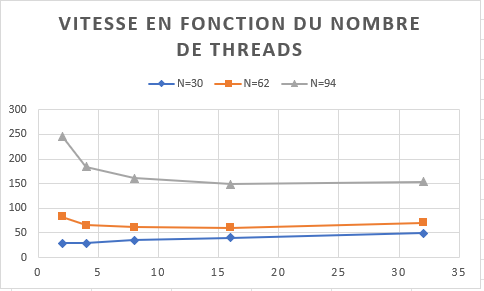
Une image contenant texte

Description générée automatiquement

Here, we once again changed a for loop for a upc\_forall loop.

Une image contenant table

Description générée automatiquementto finish it off, we print at the end the time needed to run the program for each thread.



As we can see, the performances of the program went down by a lot. A simple transcription from C to upc is not enough to get a performance boost, it’s even worse than before. This is due to the default work sharing process of the upc as it often access the memory to compute.

However, for N = 30, it is important to use the less amount of thread to keep it efficient. However for N= 62 and 94, the program runs faster the more threads there are until you pass 16 threads. After 16 threads used, the program runs slower.

**2D\_heat conduction: better memory use**

In this next exercise, we had to implement a pointer flipping in order not to copy grid\_new to grid at the end of each iteration. Thus, we are using a pointer ptr an new\_ptr pointing to grid and grid\_new. At each iteration, we are swapping the pointers ptr and new\_ptr

Une image contenant texte

Description générée automatiquement

We first had to add those pointers in the program in order to use them later.

Une image contenant texte

Description générée automatiquement

We can then initialize grid and new grid, and then initialize the pointers.

Then we just have to change the computation of in the grid by using the pointers that we created earlier.

Une image contenant texte

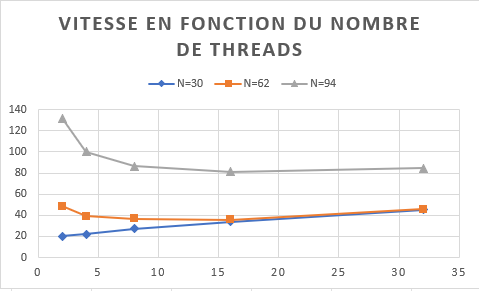
Description générée automatiquementUne image contenant texte

Description générée automatiquement

To finish the iteration, we are changing the else statement present in the heat\_1. We are now swapping the pointers before going into the next iteration.

We are here simulating a copy of grid into new grid, but it’s way faster like this.

With those pointers and pointers flipping, we are reducing by a lot the amount of synchronisation needed in the previous exercises.

Une image contenant table

Description générée automatiquement

By working with pointers, we can see almost the same progression of curve, but we here lowered all the time/iteration values. The conclusion is the same as the previous exercise: N=30 -> low number of threads, N = 62 and 94 -> 16 threads is optimal.

2D\_heat conduction: Performance boost using privatization

It was now asked to get performance boosts by using privatization of the arrays.

First of all, we are now defining 2 new constants which will be helpful later on.



Then we can add the 2 new private pointers and using malloc to allocate the memory of LOCALSIZE, which is the size needed for an array of this size.

Une image contenant texte, orange, capture d’écran

Description générée automatiquement

We are also now using upc\_memget to copy grid and new grind into the private memory - represented by our two new pointers – to initialize them.

Une image contenant texte, personne

Description générée automatiquement

Une image contenant texte

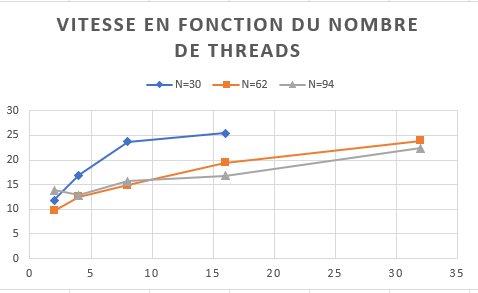
Description générée automatiquementWe should also break the upc\_forall statement into 3 blocks of operation as asked in the exercise. We will use the private pointers during the computation to benefit high performances boosts

After going through the calculation, the program will store the private array into the new\_grid array using upc\_memput. We are then synchronizing dTmax using the bupc\_allv\_reduce\_all function like before.

Une image contenant texte

Description générée automatiquement

To finish the iteration , we swap ptr and new ptr like before, but we also have to swap ptr\_priv and new\_ptr\_priv.

Une image contenant table

Description générée automatiquement

As we can see, the performance boost is huge. However, it seems like adding more and more threads will resolve in a worse performance output, unless for N=94 where 4 threads is the more efficient.

Also, during the launch of the N=30 with 32 threads program, mesoshared crashed multiple times. I couldn’t get the values of it, but it seems like the curve will flatten at around 27 µs.

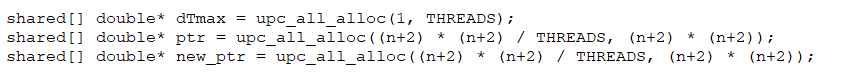
In comparison with the c code, our upc code is now running faster than the c version, unless for the N=30 where the c code is still faster. For N=62 and N=94, it is now way faster. And for the first time, the most efficient version does not belong to the N=30 but to the N=62 where we went under the 10µs.

**2D\_heat conduction: Dynamic problem size**

Here, we had to change the program to specify N at the moment of execution. Since it was too hard for me to adapt heat\_4 into heat\_5, I started over at the heat\_3 for this exercise.

Une image contenant texte

Description générée automatiquementFirst of all, we are changing the initialize function since we are not using a constant N. we also have to change the array accesses like shown through the whole code.

Also, we are redefining dTmax, ptr and new\_ptr with upc\_all\_alloc as it’s a more practical way to define them here.

Une image contenant texte

Description générée automatiquement

We also need to change the computation a bit to adapt with the new accesses of the array. We are also adding this new function, which was given on moodle, and replacing the old bupc\_allv\_reduce\_all for this one.

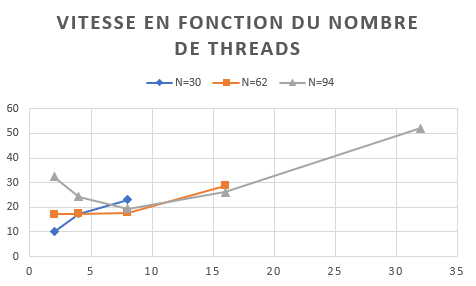
Une image contenant texte

Description générée automatiquement

To finish the iteration, like in heat\_3 and 4, we swap ptr and new\_ptr.

Une image contenant table

Description générée automatiquement

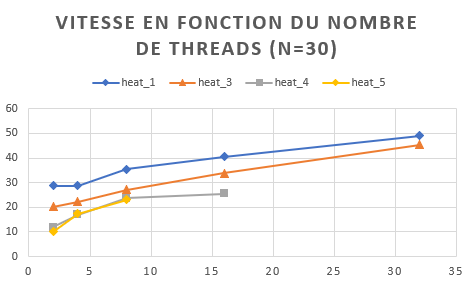


As we went back to heat\_3 to do this new exercise, the performances are a bit worst than the heat\_4 version, but are still better than the heat\_3 version. However, we are using N=30, we should use this version as it’s the fastest for it.

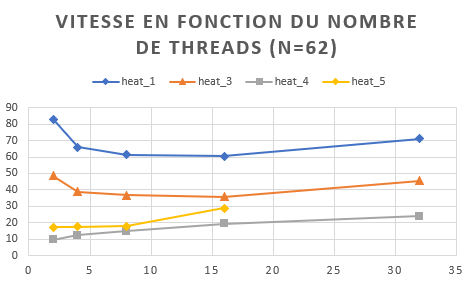
Adding more thread in the N=30 and 62 version is not optimal. For N=94, 8 threads is the most optimal.

**2D\_heat Conclusion**

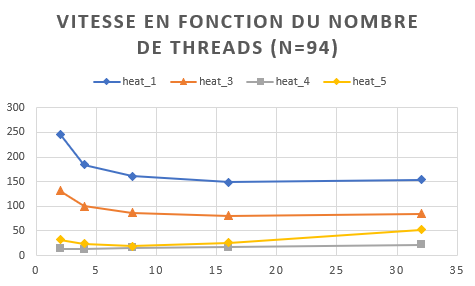
During those 2D\_heat exercises, we managed to reduce by a lot the computation time needed throughout multiple optimizations.



For N=30, we can see that the program needs to keep a low amount of threads for the computation, as the best times were the one with just 2 threads. Also, the code runs faster for each version we ran through.



For N=62, heat\_4 with the lowest number of threads is the most efficient? For the other versions (except heat\_5), 16 threads worked better.



For N=94, the computation time is the biggest out of all the other values of N (<94). The most optimal time for heat\_1 and heat\_3 is when we are using 16 threads. For heat\_5 it is 8 and it’s 4 for the heat\_4 version

At the beginning, I thought that the more threads we are using, the best the program will run. However, it’s not really working like that. In fact, it depends on a lot of factors. The computation time could be faster or lower the more threads you work with on certain versions of each code.

**Global conclusion**

Doing those exercises, I learned a lot about the upc programming and I’m now more comfortable with some concepts which were hard to understand at the beginning of the semester.

The 1D laplace solver wasn’t really a big deal since the huge part of the code was already given for each exercise? It was a nice start in the upc world.

While the 1D laplace solver didn’t offered a huge challenge, the 2D\_heat conduction part was a hard experience. I had to work a long time on it to understand all the important details and the useful concepts of upc. With a lot of time, internet searches and by speaking with my classmates, I managed to do all the exercises of the 2D\_heat conduction.