PC40: Hands on

In this report we wanted to evaluate and compare the performance of an UPC implementation and a sequential program in C on two different problems.

You will find in this report all the different versions of the code and different commentaries and explanations.

Joaquim JUSSEAU Autumn 2022

Contents

I – Simplified 1D Laplace Solver

I.1 – The 1D Solver in UPC

```
void init();

int main(int argc, char **argv){
    int j;

    init();
    upc_barrier;

    //== add a for loop which goes through the elements in the x_new array
    for( j=0; j<TOTALSIZE-1; j++ ){

        //== insert an if statement to do the work sharing across the threads
        if( j%THREADS == MYTHREAD){
            x_new[j] = 0.5*( x[j-1] + x[j+1] + b[j] );

        }
     upc_barrier;

    }


    if( MYTHREAD == 0 ){
        printf("   b   |    x    | x_new\n");
        printf("============================\n");

        for( j=0; j<TOTALSIZE; j++ )
            printf("%1.4f | %1.4f | %1.4f \n", b[j], x[j], x_new[j]);

    }

    return 0;
}

void init(){
    int i;

    if( MYTHREAD == 0 ){
        srand(time(NULL));

        for( i = 0; i<TOTALSIZE; i++ ){
            b[i] = (double)rand() / RAND_MAX;
            x[i] = (double)rand() / RAND_MAX;
        }
    }
}
```

Figure 1 - Exo_3.upc

In this first case, we add a classic for loop, followed by an if statement in order to distribute the work. The loop will sequence though all “TOTALSIZE -1” iterations, assigning an iteration to each thread in classic round-robin fashion. Actually, this implementation is clearly not efficient, indeed each thread evaluates the loop header TOTALSIZE -1 + 1 times, this redundant loop overhead may have nearly the same temporal cost as that the sequential program in C.

I.2 – Better work sharing construct with a single loop

```
#define TOTALSIZE 16

shared double x[TOTALSIZE*THREADS];
shared double x_new[TOTALSIZE*THREADS];
shared double b[TOTALSIZE*THREADS];

void init();

int main(int argc, char **argv){
    int j;

    init();
    upc_barrier;

    // ==> setup j to point to the first element so that the current thread should
    // progress (in respect to its affinity)

     j = *x_new ;



    // ==> add a for loop which goes only through the elements in the x_new array
    // with affinity to the current THREAD

    for( j=MYTHREAD; j<TOTALSIZE*THREADS; j+=THREADS )
        x_new[j] = 0.5*( x[j-1] + x[j+1] + b[j] );

    upc_barrier;

    if( MYTHREAD == 0 ){
        printf("   b   |    x    | x_new\n");
        printf("============================\n");

        for( j=0; j<TOTALSIZE*THREADS; j++ )
            printf("%1.4f | %1.4f | %1.4f \n", b[j], x[j], x_new[j]);
    }

    return 0;
}

void init(){
    int i;

    if( MYTHREAD == 0 ){
        srand(time(NULL));

        for( i=0 ; i<TOTALSIZE*THREADS; i++ ){
            b[i] = (double)rand() / RAND_MAX;
            x[i] = (double)rand() / RAND_MAX;
        }
    }
}
```

Figure 2 - Exo_4.upc

I.3 – Blocked arrays and upc_forall

In order to solve ex_3.upc issues we changed the for loop with a different initialization for j and incrementation of THREADS each time.

To remove the race condition, we can implement a upc_barrier in order to synchronize the results and ensure that all threads must reach that point before any of them can proceed further.

```
#include <upc_relaxed.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define BLOCKSIZE 16

//==> declare the x, x_new and b arrays in the shared space with size of
//    BLOCKSIZE*THREADS and with blocking size of BLOCKSIZE
shared [BLOCKSIZE] double x[BLOCKSIZE*THREADS];
shared [BLOCKSIZE] double x_new[BLOCKSIZE*THREADS];
shared [BLOCKSIZE] double b[BLOCKSIZE*THREADS];

void init();

int main(int argc, char **argv){
    int j;

    init();
    upc_barrier;
    //==> insert a upc_forall statement to do work sharing while
    //    respecting the affinity of the x_new array


    upc_forall( j=0; j<(BLOCKSIZE*THREADS)-1; j++; &x_new[j]){
        x_new[j] = 0.5*( x[j-1] + x[j+1] + b[j] );
    }
    upc_barrier;

    if( MYTHREAD == 0 ){
        printf("   b   |    x    | x_new\n");
        printf("============================\n");

        for( j=0; j<BLOCKSIZE*THREADS; j++ )
            printf("%1.4f | %1.4f | %1.4f \n", b[j], x[j], x_new[j]);
    }

    return 0;
}

void init(){
    int i;

    if( MYTHREAD == 0 ){
        srand(time(NULL));

        for( i=0; i<BLOCKSIZE*THREADS; i++ ){
            b[i] = (double)rand() / RAND_MAX;
            x[i] = (double)rand() / RAND_MAX;
        }
    }
}
```

Figure 3 - Exo_5.upc

To conclude the improvement of the loop, we use an upc_forall construct to distribute the work, in this case, the affinity &x_new[j] indicates that iteration j will respect the affinity of the x_new array.

I.4 - Synchronization

```
shared [BLOCKSIZE] double x[BLOCKSIZE*THREADS];
shared [BLOCKSIZE] double x_new[BLOCKSIZE*THREADS];
shared [BLOCKSIZE] double b[BLOCKSIZE*THREADS];

void init();

int main(int argc, char **argv){
    int j;
    int iter;

    init();
    upc_barrier;

    // add two barrier statements, to ensure all threads finished computing
    // x_new[] and to ensure that all threads have completed the array
    // swapping.
    for( iter=0; iter<10000; iter++ ){
        upc_forall( j=1; j<BLOCKSIZE*THREADS-1; j++; &x_new[j] ){
            x_new[j] = 0.5*( x[j-1] + x[j+1] + b[j] );
        }
          upc_barrier;

        upc_forall( j=0; j<BLOCKSIZE*THREADS; j++; &x_new[j] ){
            x[j] = x_new[j];
        }
          upc_barrier;
    }

    if( MYTHREAD == 0 ){
        printf("   b   |    x    | x_new\n");
        printf("============================\n");

        for( j=0; j<BLOCKSIZE*THREADS; j++ )
            printf("%1.4f | %1.4f | %1.4f \n", b[j], x[j], x_new[j]);
    }

    return 0;
}

void init(){
    int i;

    if( MYTHREAD == 0 ){
        srand(time(NULL));

        for( i=0; i<BLOCKSIZE*THREADS; i++ ){
            b[i] = (double)rand() / RAND_MAX;
            x[i] = (double)rand() / RAND_MAX;
        }
    }
}
```

Figure 4 - Ex_6.upc

It is important to note that we have to use barrier synchronization after each upc_forall statement, this is because UPC specification does not require an implicit barrier at the end of the iteration statement.

I.5 - Reduction operation

```
shared [TOTALSIZE] double x[TOTALSIZE*THREADS];
shared [TOTALSIZE] double x_new[TOTALSIZE*THREADS];
shared [TOTALSIZE] double b[TOTALSIZE*THREADS];
shared double diff[THREADS];
shared double diffmax;

void init(){
    int i;

    for( i = 0; i < TOTALSIZE*THREADS; i++ ){
        b[i] = 0;
        x[i] = 0;
    }

    b[1] = 1.0;
    b[TOTALSIZE*THREADS-2] = 1.0;
}

int main(){
    int j;
    int iter = 0;

    if( MYTHREAD == 0 )
        init();

    upc_barrier;

    while( 1 ){
        iter++;
        diff[MYTHREAD] = 0.0;

        upc_forall( j=1; j<TOTALSIZE*THREADS-1; j++; &x_new[j] ){
            x_new[j] = 0.5 * ( x[j-1] + x[j+1] + b[j] );

            if( diff[MYTHREAD] < x_new[j] - x[j] )
                diff[MYTHREAD] = x_new[j] - x[j];
        }

        // Each thread as a local value for diff
        // The maximum of those values should be used to check
        // the convergence.


        upc_all_reduceD(&diffmax,diff,UPC_MAX,THREADS,1,NULL,UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);

        printf("diff max = %f \n", diffmax);

        if( diffmax <= EPSILON )
            break;
        if( iter > 10000 )
            break;

        upc_forall( j=0; j<TOTALSIZE*THREADS; j++; &x_new[j] ){
            x[j] = x_new[j];
        }
        upc_barrier;
    }
```

Figure 5 - Ex_7.upc

Here, we use the upc_all_reduce construct in order to compute the global difference diffmax, we can see that we have the letter D after upc_all_reduce, signifying the double type.

II - 2D Heat conduction

II.1 First UPC program

```
#include <upc.h>
#include <stdio.h>
#include <math.h>

#define N 798
#define BLOCKSIZE ((N+2) * (N+2)/THREADS)

shared[BLOCKSIZE] double grid [N+2][N+2] ;
shared[BLOCKSIZE] double new_grid [N+2][N+2];
shared[BLOCKSIZE] double dTmax_local[THREADS];

void initialize(void)
{
    int j;

    /* Heat one side of the solid */
    for( j=1; j<N+1; j++ )
    {
        grid[0][j] = 1.0;
        new_grid[0][j] = 1.0;
    }
}
```

```
int main(void)
{
    struct timeval ts_st, ts_end;
    double dTmax, dT, epsilon, time;
    int finished, i, j, k, l;
    double T;
    int nr_iter;

      initialize();

    epsilon  = 0.0001;
    finished = 0;
    nr_iter = 0;

    upc_barrier; // SYNCHRONIZATION


    /* and start the timed section */
    gettimeofday( &ts_st, NULL );

    do
    {
        dTmax = 0.0;
        for( i=1; i<N+1; i++ )
        {
            upc_forall( j=1; j<N+1; j++;&grid[i][j] )
            {
                T = 0.25 *
                    (grid[i+1][j] + grid[i-1][j] +
                     grid[i][j-1] + grid[i][j+1]); /* stencil */
                dT = T - grid[i][j]; /* local variation */
                new_grid[i][j] = T;
                if( dTmax < fabs(dT) )
                    dTmax = fabs(dT); /* max variation in this iteration */
            }
        }

        dTmax_local[MYTHREAD] = dTmax;
        upc_barrier;

        dTmax = dTmax_local[0];
        for(i=1;i<THREADS;i++)
            if(dTmax < dTmax_local[i])
                  dTmax = dTmax_local[i];

        upc_barrier ;

        if( dTmax < epsilon ) /* is the precision reached good enough ? */
            finished = 1;
        else
        {
            for( k=0; k<N+2; k++ )      /* not yet ... Need to prepare */
                upc_forall( l=0; l<N+2; l++ ;&grid[k][l])    /* ourselves for doing a new */
                    grid[k][l] = new_grid[k][l]; /* iteration */
        }
        nr_iter++;
    } while( finished == 0 );

    gettimeofday( &ts_end, NULL ); /* end the timed section */

    /* compute the execution time */
    time = ts_end.tv_sec + (ts_end.tv_usec / 1000000.0);
    time -= ts_st.tv_sec + (ts_st.tv_usec / 1000000.0);

    printf("%d iterations in %.3lf sec\n", nr_iter, time);

    return 0;
```

For this first implementation in UPC, based on the sequential program in C, we added and replaced one for loop by an upc_forall statement. We can notice that we have “&grid[i][j]” in the affinity field instead of a classic number j, indeed each thread will execute a given iteration if the object to has affinity to that thread, if the object pointed to was local to that thread.

We can see that we added a shared array called dTmax_local[THREADS], that has one element per thread to hold the local maximum temperature, on top of that it is important to notice the upc_barrier which is applied to ensure that all threads have deposited their local maximum temperature change into that array before a global maximum is computed. The next barrier synchronization ensures

that all threads have gone through the shared array dTmax_local[THREADS] and determined the maximum value, so the next iteration can reuse the array buffer.

Concerning the performance of this first program it is clearly not the best because of a lot of remote access and no privatization. Moreover, 8 threads seem to be the best choice.

II.2 - Better memory use

```
#define N 798

shared double grid [N+2][N+2] ;
shared double new_grid [N+2][N+2];
shared double dTmax_local[THREADS];

shared double *ptr[N+2];
shared double *new_ptr[N+2];

//shared double dTmax ;


void initialize(void)
{
    int j;

    /* Heat one side of the solid */
    for( j=1; j<N+1; j++ )
    {
        grid[0][j] = 1.0;
        new_grid[0][j] = 1.0;
    }
    int i;
    for(i=0;i<N+2;i++)
    {
        ptr[i] = &grid[i][0];
        new_ptr[i] = &new_grid[i][0];

    }
}
```

```
int main(void)
{
    struct timeval ts_st, ts_end;


    double dTmax ;
    double dT, epsilon, time;
    int finished, i, j, k, l;
    double T;
    int nr_iter;

    initialize();

    epsilon  = 0.0001;
    finished = 0;
    nr_iter = 0;

    upc_barrier; // SYNCHRONIZATION


    /* and start the timed section */
    gettimeofday( &ts_st, NULL );

    do
    {
        dTmax = 0.0;
        for( i=1; i<N+1; i++ )
        {
            upc_forall( j=1; j<N+1; j++;&grid[i][j] )
            {T = 0.25 *
                    (ptr[i+1][j] + ptr[i-1][j] +
                    ptr[i][j-1] + ptr[i][j+1]); /* stencil */
                dT = T - ptr[i][j]; /* local variation */
                new_ptr[i][j] = T;
                if( dTmax < fabs(dT) )
                    dTmax = fabs(dT); /* max variation in this iteration */
            }
        }

        dTmax_local[MYTHREAD] = dTmax;
        upc_barrier;


        dTmax = dTmax_local[0];
        for(i=1;i<THREADS;i++)
                if(dTmax < dTmax_local[i])
                        dTmax = dTmax_local[i];


        //upc_all_reduceD(&dTmax,dTmax_local,UPC_MAX,THREADS,1,NULL,UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);

        upc_barrier ;
```

```
        if( dTmax < epsilon ) /* is the precision reached good enough ? */
            finished = 1;
        else
        {
            for( k=0; k<N+2; k++ )      /* not yet ... Need to prepare */
                for( l=0; l<N+2; l++ )    /* ourselves for doing a new */
                    ptr[k][l] = new_ptr[k][l]; /* iteration */
        }
        nr_iter++;
    } while( finished == 0 );

    gettimeofday( &ts_end, NULL ); /* end the timed section */

    /* compute the execution time */
    time = ts_end.tv_sec + (ts_end.tv_usec / 1000000.0);
    time -= ts_st.tv_sec + (ts_st.tv_usec / 1000000.0);

    printf("%d iterations in %.3lf sec\n", nr_iter, time);

    return 0;
```

For this exercise, we added several interesting things. First of all, we declared two new arrays of shared pointers, in order to optimize the stencil computation and to avoid overhead in the copying step by implementing a pointer flipping. We now have a complexity of O(1) instead of O(n) at the end of each iteration. We can already see on the curve below that this program is more efficient than the previous with just two shared arrays. Concerning the computation of dTmax we can also notice that we tried to implement an upc_all_reduce, it is efficient but not enough to be significant. On top of that if we want to implement this construct, we have to declare a global variable ‘shared double dTmax’.

II.3 Performance boost using privatization

Here, the main thing was to understand how privatization work and how to divide the previous upc_forall statement in three different blocks. In order to make less shared accesses, our program can copy a chunk of the grid array directly into the private memory using the function upc_memget, now we can operate on the private array and just use remote accesses, when necessary, when all the tasks are finished, we put the private array into new_grid thanks to the function upc_memput.

```
#include <upc.h>
#include <stdio.h>
#include <math.h>

#define N 798
#define BLOCKSIZE ( (N+2) * (N+2) / THREADS)
#define LOCALW ((N+2)/THREADS)
#define LOCALS (LOCALW * sizeof(double) * (N+2))

shared[BLOCKSIZE] double grid [N+2][N+2];
shared[BLOCKSIZE] double new_grid [N+2][N+2];
shared double dTmax_local[THREADS];

shared[BLOCKSIZE] double *ptr[N+2];
shared[BLOCKSIZE] double *new_ptr[N+2];


void initialize(void)
{
    int j;

    /* Heat one side of the solid */
    for( j=1; j<N+1; j++ )
    {
        grid[0][j] = 1.0;
        new_grid[0][j] = 1.0;
    }


    //ptr_priv_J[0] = (double(*)[N+2][N+2]) &grid[N*MYTHREAD/THREADS][0];

     int global_z = n*MYTHREAD/THREADS;



     int i;
    for(i=0;i<N+2;i++)
    {
     ptr[i] = &grid[i][0];
        new_ptr[i] = &new_grid[i][0];


    }


}
```

```
int main(void)
{



     double *ptr_priv[N+2] = malloc(LOCALS) ;
     double *new_ptr_priv[N+2]=malloc(LOCALS);

    struct timeval ts_st, ts_end;
    double dTmax, dT, epsilon, time;
    int finished, i, j, k, l;
    double T;
    int nr_iter;

     initialize();

    epsilon  = 0.0001;
    finished = 0;
    nr_iter = 0;

    upc_barrier; // SYNCHRONIZATION

     upc_memget(ptr_priv,&grid[LOCALW * MYTHREAD],LOCALS);
     upc_memget(new_ptr_priv,&new_grid[LOCALW * MYTHREAD],LOCALS);


    /* and start the timed section */
    gettimeofday( &ts_st, NULL );

    do
    {
        dTmax = 0.0;

         int o = LOCALW * MYTHREAD;
         int k = 0;
         if(i+o >0){

              for(int k =1; k<=N ; k++){

                   T=0.25* ptr_priv [i+1][k] + ptr [ o+i-1 ][ k ] + ptr_priv [i][k-1] + ptr_priv [i][k+1]);
                   dT = fabs(T - ptr_priv[i][k];
                   new_ptr_priv[i][k]= T;
                   if(dTmax<dT) dTmax = dT;


              }

         }
         for ( i +=1 ;i<LOCALW -1 ;i++){
              for( int k=1; k<=N; k++){
                   T=0.25* ptr_priv [i+1][k] + ptr [ o+i-1 ][ k ] + ptr_priv [i][k-1] + ptr_priv [i][k+1]);
                   dT = fabs(T - ptr_priv[i][k];
                   new_ptr_priv[i][k]= T;
                   if(dTmax<dT) dTmax = dT;
              }
```

```
         }
         if (i+o <N+1){
              for(int k =1; k<=N ; k++){

                   T=0.25 * (ptr[o+i+1][j] + ptr_priv[i-1][j] + ptr_priv[i][j-1] + ptr_priv[i][j+1]) ;
                   dT = fabs(T - ptr_priv[i][k];
                   new_ptr_priv[i][k]= T;
                   if(dTmax<dT) dTmax = dT;
              }
         }
         upc_memput(&new_ptr[LOCALW * MYTHREAD],new_ptr_priv,LOCALS);

         double dTmax_global = bupc_allv_reduce_all(double, dTmax, UPC_MAX);


         if(dTmax_global < EPSILON){
              finished = 1;

         }else{
              shared[BLOCKSIZE] double (*temp_ptr)[N+2] = ptr;
              ptr= new_ptr;
              new_ptr = temp_ptr;

              double (*temp_ptr_priv)[N+2] = ptr_priv;
              ptr_priv = new_ptr_priv;
              new_ptr_priv = temp_ptr_priv;
         }

        nr_iter++;
    } while( finished == 0 );

    gettimeofday( &ts_end, NULL ); /* end the timed section */

    /* compute the execution time */
    time = ts_end.tv_sec + (ts_end.tv_usec / 1000000.0);
    time -= ts_st.tv_sec + (ts_st.tv_usec / 1000000.0);

    printf("%d iterations in %.3lf sec\n", nr_iter, time);

    return 0;
```

II.4 Dynamic Problem Size

This is the last version of our problem; the big point is to transform the previous one to a dynamic program, in other words we will specify the number of threads and the size of the grid at runtime. We made a lot of modification; we can notice that we deleted the declaration of grid and new_grid because it is now done implicitly trough the function upc_all_alloc.

```
#include <upc.h>
#include <stdio.h>
#include <math.h>
#include <upc_collective.h>
#include <math.h>



shared double dTmax_g;
void initialize(shared[] double* grid, shared[] double* new_grid, int n){
     for(int i = 0; i<n+2;i++){
          for(int j=0;j<n+2;j++){
               grid[i*(n+2) +j] =0.0;
               new_grid[i*(n+2) +j] =0.0;

          }
     }

}

int main(int argc, char* argv[]) {
    if (argc != 2) {
        if (MYTHREAD == 0) fprintf(stderr, "Error: expected two arguments, got %d\n", argc);
        exit(1);
    }
    struct timeval ts_st, ts_end;
    double dT, EPSILON , time;
    int finished, i, j, k, l;
    double T;

    int nr_iter;



    int n = atoi(argv[1]);

    shared[] double* dTmax = upc_all_alloc(1, THREADS);
    shared[] double* ptr = upc_all_alloc((n+2) * (n+2) / THREADS, (n+2) * (n+2));
    shared[] double* new_ptr = upc_all_alloc((n+2) * (n+2) / THREADS, (n+2) * (n+2));


    #define ptr_get(x, y) ptr[(x) * (n+2) + (y)]

    if (MYTHREAD == 0) {
        initialize(ptr, new_ptr, n);
    }
    EPSILON = 0.0001;
    upc_barrier;
    finished = 0;
    int n_iter = 0;
```

```
    gettimeofday( &ts_st, NULL );
    do {
        dTmax[MYTHREAD] = 0.0;
        size_t i = (n+2) * MYTHREAD / THREADS;
        if (i == 0) i = 1;
        size_t imax = (n+2) * (MYTHREAD + 1) / THREADS;
        if (imax > n+1) imax = n+1;

        for (; i < imax; i++) {
            for (int j = 1; j <= n; j++) {
                 T = 0.25 * (ptr_get(i+1, j) + ptr_get(i-1, j) + ptr_get(i, j-1) + ptr_get(i, j+1));
                 dT = fabs(T - ptr_get(i, j));
                new_ptr[i * (n+2) + j] = T;
                if (dTmax[MYTHREAD] < dT) dTmax[MYTHREAD] = dT;
            }
        }

        upc_all_reduceD(&dTmax_g, dTmax, UPC_MAX, THREADS, 1, NULL, UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);

        if (dTmax_g < EPSILON) {
            finished = 1;
        } else {
            shared[] double* tmp = ptr;
            ptr = new_ptr;
            new_ptr = tmp;
        }
        n_iter++;
        // upc_barrier;
    } while (finished=1);
    gettimeofday( &ts_end, NULL ); /* end the timed section */

    /* compute the execution time */
    time = ts_end.tv_sec + (ts_end.tv_usec / 1000000.0);
    time -= ts_st.tv_sec + (ts_st.tv_usec / 1000000.0);

    printf("%d iterations in %.3lf sec\n", nr_iter, time);

    return 0;
}
```

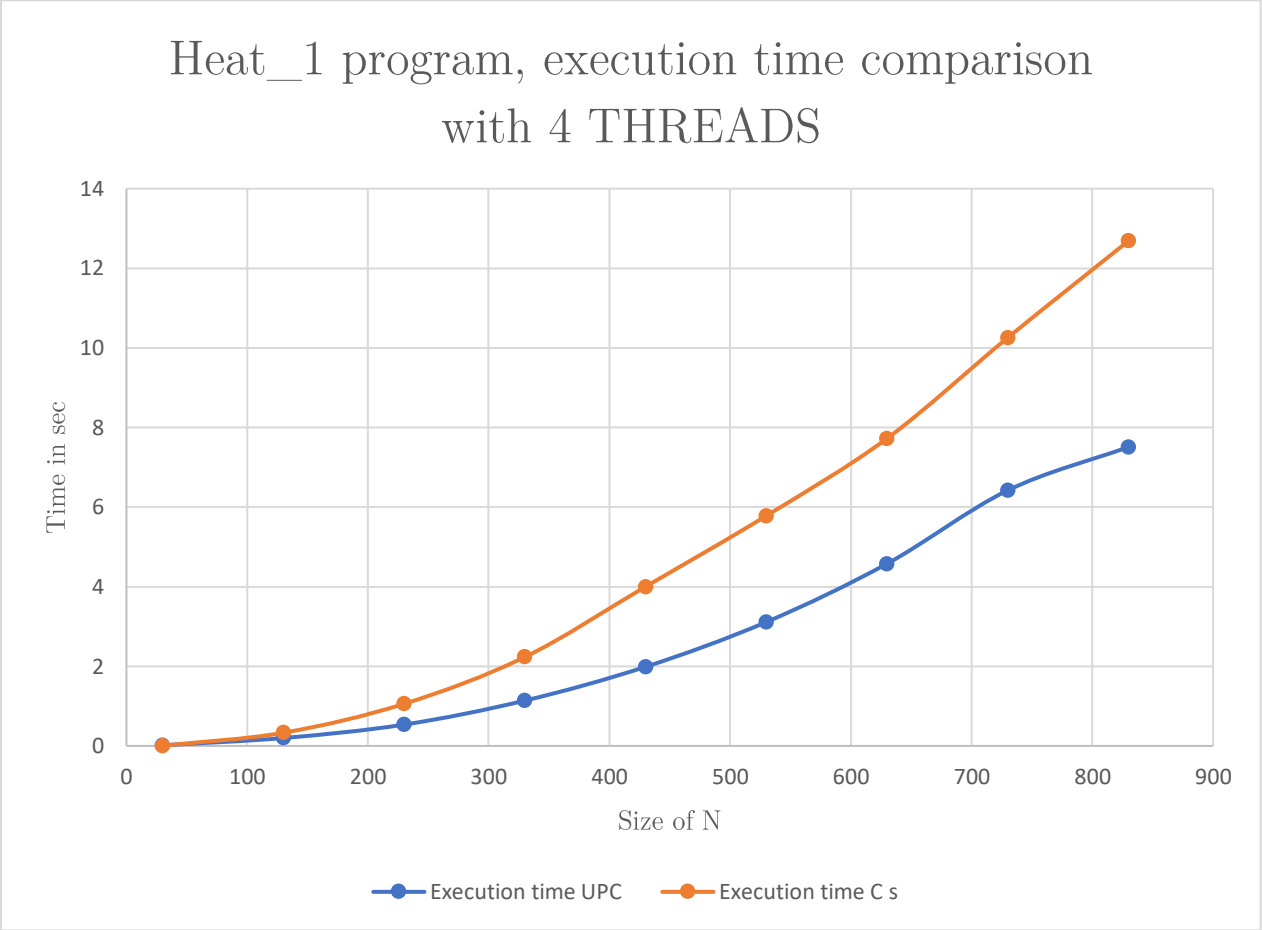
Conclusion:

The first exercise (Simplified Laplace Equation Solver) was quite easily parallelizable, actually the second exercise was harder specially concerning the privatization of the data. The main thing was to minimalize the remote accesses and optimize local and private connection.

Despite all my effort, the UPC implementation is not directly the best and the most efficient compared to the sequential implementation, I was surprised to see how difficult it is to beat the C program, but privatization seems to be a great way to improve our execution time and our efficiency.

Excel speed up analysis:

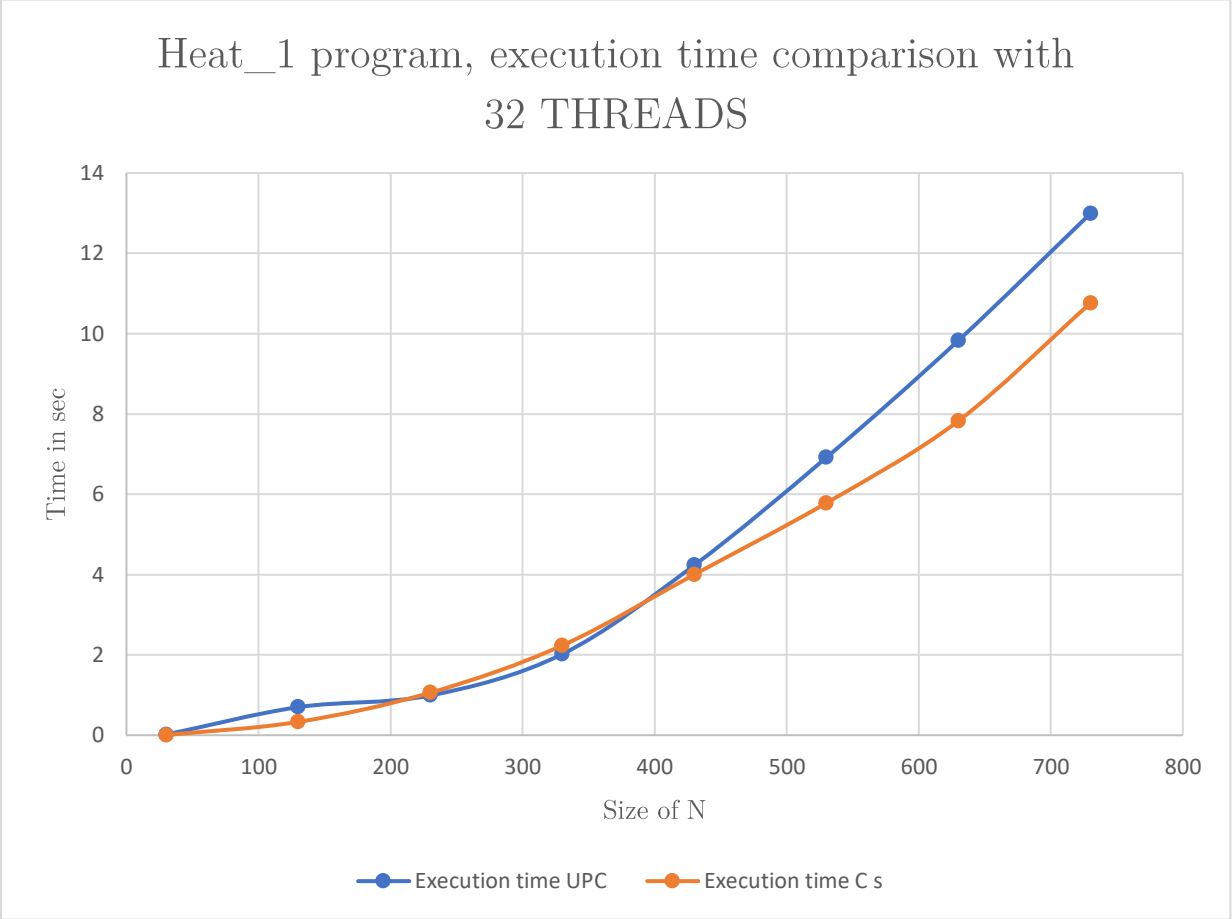
Before explanation, it is essential to say that all the programs were tested on Mesoshared and not on a local computer, and all the programs were not tested at the same time at the same place, so a lot of factors can modify execution time. However, all the curves are representative of the efficiency of UPC and help in the comprehension of the exercises and show the best implementation



SIZE N :	30	130	230	330	430	530	630	730	830
Execution time UPC	0,017	0,199	0,536	1,139	1,988	3,113	4,578	6,421	7,51
Execution time C s	0,005	0,332	1,058	2,234	4,002	5,782	7,725	10,257	12,689

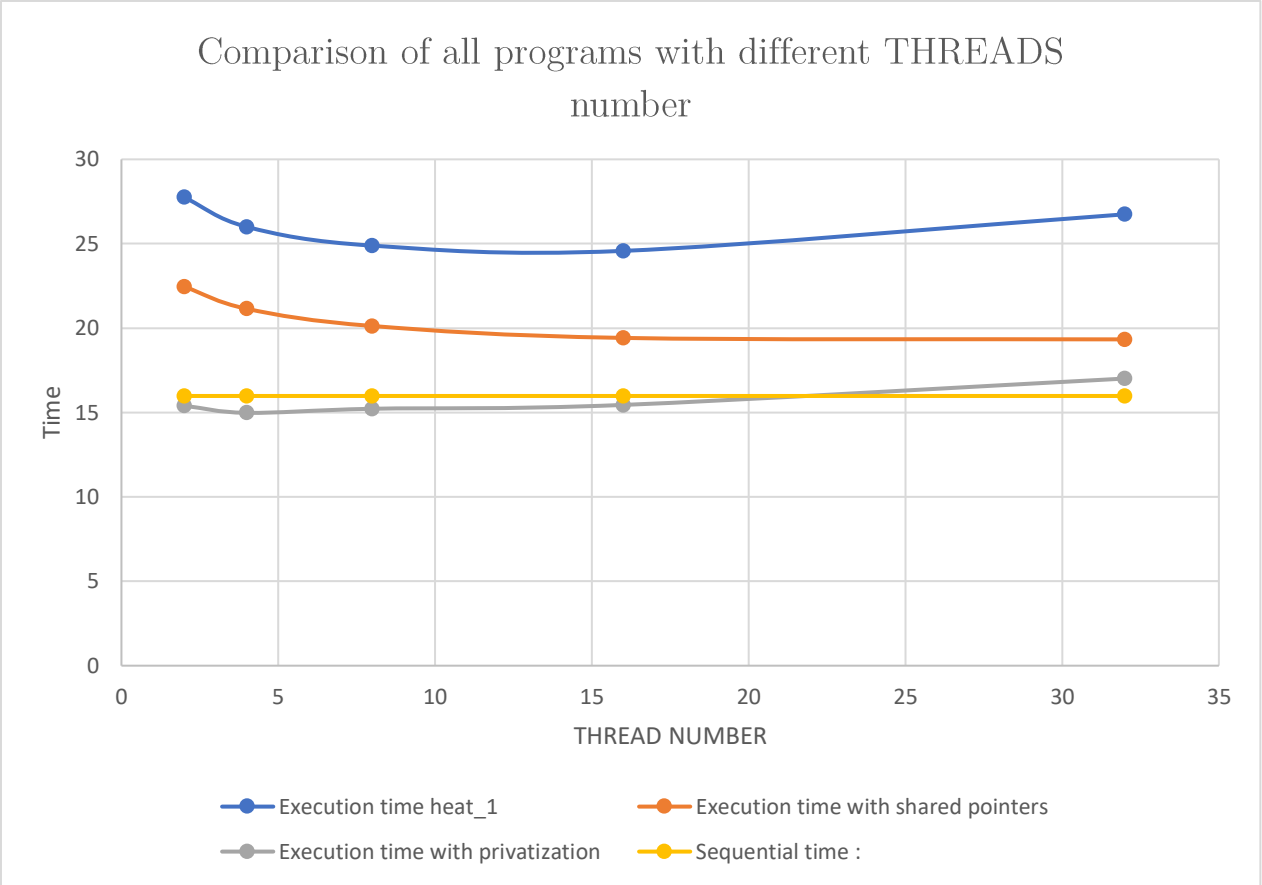
In this first graphic, we cans see the difference of the parallel and sequential program when the size of N increases. When N is between 100 and 330, we can see that the difference is not huge but when we are over 500, we have 2 seconds

of difference, and this keep increasing. We can also notice the fact that the execution time of UPC begin to stabilize around a size N of 830.



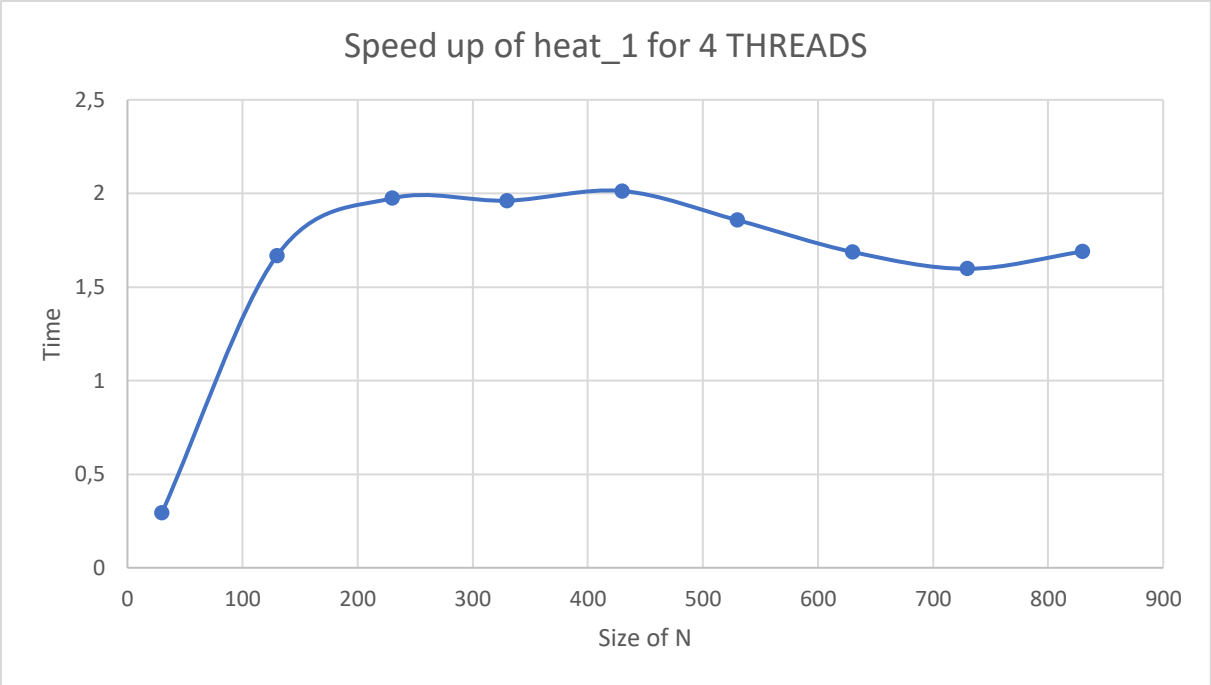
SIZE N :	30	130	230	330	430	530	630	730
Execution time UPC	0,017	0,499	0,936	1,939	3,989	6,113	8,428	11,587
Execution time C s	0,005	0,332	1,058	2,234	4,002	5,782	7,825	10,757

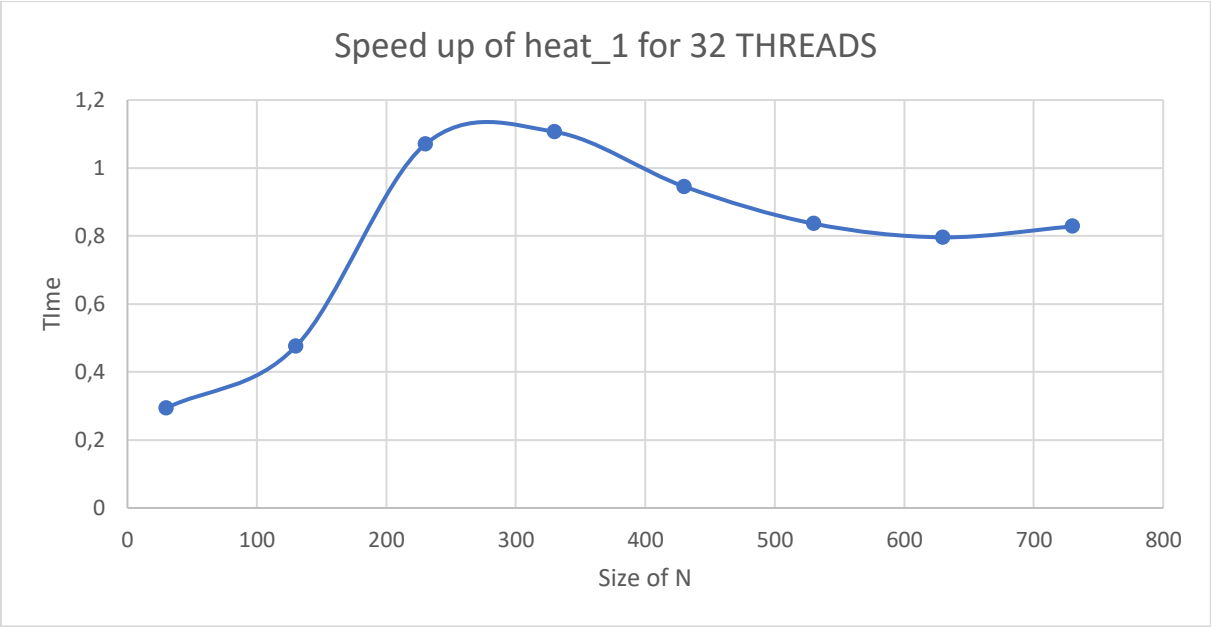
Logically, if we increase too much the threads number the UPC implementation became worst than the sequential program, so parallelization is not efficient et not useful.



Thread number :	2	4	8	16	32
Execution time heat_1	27,752	26	24,889	24,577	26,745
Execution time with shared pointers	22,457	21,145	20,127	19,42	19,333
Execution time with privatization	15,4	14,981	15,222	15,449	17,012
Sequential time :	15,98	15,98	15,98	15,98	15,98

This is the most interesting part; indeed, we clearly see how our modification on our program can impact the execution time. The most efficient is logically the program with implementation of privatize pointers but we can also see that if we reach and try an execution with 32 threads, the C implementation became better and most efficient.





Speed up curves is also an excellent way to evaluate the performance of our programs, indeed, we can clearly notice that the efficacity of heat_1 with 4 Threads is better with a maximum of 2,1 versus 1.7 for the 32 THREADS programs

References

1 -UPC : Distributed Shared-Memory Programming

by Tarek El-Ghazawi (Author), William Carlson (Author), Thomas Sterling (Author), Katherine Yelick (Author)

2 -UPC : UPC Collective Operations Specifications
V1.0

A publication of the UPC Consortium, December 12,2003