PC40 Hands-on: UPC

Eileen Jovenin INFO01

Autumn 2022

This report is the discovering of UPC through practicum on computer. It combines all the codes with some commentaries and analysis.
Every code has been run on the `mesoshared` server, which limited the number of threads and so the speed of the program.

Contents

1 Simplified 1D Laplace solver

1.1 C implementation

The C implementation of the Laplace solver has only one thread contrary to the .upc versions.

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

#define TOTALSIZE 800

void init();
double x_new[TOTALSIZE];
double x[TOTALSIZE];
double b[TOTALSIZE];

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

    init();

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

    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;

    srand( time(NULL) );

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

Listing 1: C implementation of the 1D Laplace solver

1.2 The 1D solver in UPC

This code was mainly already wrote, despite some parts which I have to fill. It was designed to be able to quickly port it to UPC.
A new `if` statement had to be inserted in the `for` loop within `iteration()`. Additionally, several lines had to only be executed by the thread 0.

The `x`, `xnew` and `b` arrays were made shared and `upc_barrier` statements were added at the end of `init()`, `iteration()` and `copy_array()` to prevent threads from beginning processing the next iteration when the `x` and `xnew` arrays aren't ready, and to prevent the thread 0 from stopping too early and printing the wrong timing.

This code yields the following results:

Threads	Time
2	$512.5\mu s$/iter
3	$539.4\mu s$/iter
4	$334.9\mu s$/iter
8	$212.9\mu s$/iter
16	$159.7\mu s$/iter
32	$148.1\mu s$/iter

Table 1: Timing results for the first UPC implementation (Listing 2)

When compiled and run with 3 threads, the code runs noticeably slower.

Threads = 24, Time = $252.6\mu s$/iter (expected ≈ 160 μs/iter if I compare with the Table 1)
Threads = 31, Time = $317.6\mu s$/iter (expected ≈ 150 μs/iter if I compare with the Table 1)

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

#define TOTALSIZE       800

//== declare the x, x_new, b arrays in the shared space with size of TOTALSIZE
shared double x[TOTALSIZE];
shared double x_new[TOTALSIZE];
shared double b[TOTALSIZE];

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] );
        }
    }

    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;
        }
    }
}
```

Listing 2: First UPC implementation of the 1D Laplace solver

1.3 Better work sharing construct with a single for loop

The first step in optimizing the UPC implementation of the 1D Laplace equation solver is to replace the `for (...) if (...)` with a single `for`.

This change increases the speed of the program:

Threads	Time
2	$480.2\mu s/\text{iter}$
4	$268.7\mu s/\text{iter}$
8	$126.5\mu s/\text{iter}$
16	$73.3\mu s/\text{iter}$
32	$41.9\mu s/\text{iter}$

Table 2: Timing results for the second UPC implementation of the 1D Laplace solver

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

#define TOTALSIZE 32

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)
```

```c

    // ==> 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;
        }
    }
}
```

Listing 3: Second UPC implementation of the 1D Laplace solver

1.4 Blocked arrays and Work sharing with upc forall

Our next optimization is to increase the blocking factor N for the x, x_{new} and b arrays. Each operation on the item of index j accesses $b[j]$, $x[j-1]$, $x[j+1]$ and $x_{new}[j]$.

With the default block factor of 1, accesses to both $x[j-1]$ and $x[j+1]$ will always be outside the current thread's affinity. That's why a greater block factor can be made, which will reduce to only $2/N$ accesses on average.

The effects of this change depend on N: the code runs fastest when $N = 2^n$. This can be seen in the Table 1.

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

#define TOTALSIZE 32

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

```c

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<(TOTALSIZE*THREADS)-1; j++; 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<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;
        }
    }
}
```

Listing 4: ex5.upc

Threads	Time
2	$957.5\mu s/\text{iter}$
4	$862.2\mu s/\text{iter}$
8	$690.7\mu s/\text{iter}$
16	$580.0\mu s/\text{iter}$
32	$532.7\mu s/\text{iter}$

Table 3: Timing results for the third UPC implementation of the 1D Laplace solver, $N = 32$

1.5 Synchronization

```c
#include <upc_relaxed.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define TOTALSIZE 32

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

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

    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;
        }
    }
}
```

Listing 5: ex6.upc

1.6 Reduction operation

One of my last modifications is to measure $\delta_{max} = \max_{i\in[\![0;n-1]\!]} |x[i] - x_{new}[i]|$ (named `diffmax` in the source code) and to stop once $\delta_{max} \leqslant \varepsilon$.

The new timings are:

Threads	Time
2	$1402.3\mu s$/iter
4	$1098.4\mu s$/iter
8	$804.8\mu s$/iter
16	$643.6\mu s$/iter
32	$570.1\mu s$/iter

Table 4: Timing results for the fourth UPC implementation of the 1D Laplace solver with $N = 32$

```
#include <upc.h>
// #include <upc_collective.h>   It is recommended that you use
//             the collectives for this exercise...
#include <stdio.h>
#include <math.h>

#define TOTALSIZE 100
#define EPSILON 0.000001

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.

        // diffmax = max(diff[0..THREADS - 1])

        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;
    }

    /* You can display the results here :
    if( MYTHREAD == 0 ){
        for(j=0; j<TOTALSIZE*THREADS; j++){
            printf("%f\t", x_new[j]);
        }
        printf("\n");
    }
    */

    return 0;
}
```

Listing 6: ex7.upc

1.7 Conclusion

The first implementation in UPC and its subsequent first optimization gave very good results for the improvement of the speed of the program, decreasing the timings by almost $100\mu s$/iter.

Unfortunately, the second optimization attempt gave higher timings and made the UPC implementation slower than the C implementation, which is not wanted.

The resulting program (with δ_{max}) runs slightly faster than the original C implementation, given enough threads, but the first optimization is still the best in terms of speed.

2 2D Heat conduction

As with Section 1, I begin with a simple C implementation of the algorithm.

2.1 Sequential C program

The performance result for the first C implementation, ran on `mesoshared` is an average of $89.1\mu s/\text{iter}$.

```c
#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#define N 498

double grid[N+2][N+2], new_grid[N+2][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;
    }
}

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();

    /* Set the precision wanted */
    epsilon  = 0.0001;
    finished = 0;
    nr_iter = 0;
    /* and start the timed section */
    gettimeofday( &ts_st, NULL );

    do
    {
        dTmax = 0.0;
        for( i=1; i<N+1; i++ )
        {
            for( j=1; j<N+1; 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 */
                }
            }
            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 */
                        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;
}
```

Listing 7: C implementation of the 2D Heat simulation (heat_c)

2.2 First UPC program

Creating the first implementation in UPC is apparent. However the speed are severely decreased due to work sharing which causes a lot of remote accesses to memory. I obtain the following measurements on `mesoshared`:

Threads	Time
2	258.2μs/iter
4	192.3μs/iter
8	162.2μs/iter
16	150.5μs/iter
32	152.6μs/iter

Table 5: Timing results for `heat_1.upc`

```
#include <upc_relaxed.h>
#include <bupc_collectivev.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdbool.h>

#define N 94
#define EPSILON 0.0001
#define TOTALSIZE ((N+2) * (N+2) / THREADS)

shared[TOTALSIZE] double grid[N+2][N+2];
shared[TOTALSIZE] double new_grid[N+2][N+2];
shared double dTmax[THREADS];

```

```
void init() {
    for (size_t j = 1; j < N+1; j++) {
        grid[0][j] = 1.0;
        new_grid[0][j] = 1.0;
    }
}

int main() {
    if (MYTHREAD == 0) {
        init();
    }

    upc_barrier;
    bool finished = false;
    int n_iter = 0;

    clock_t begin = clock();
    do {
        double dTmax = 0.0;
        for (size_t i = 1; i <= N; i++) {
            upc_forall (size_t j = 1; j <= N; j++; &grid[i][j]) {
                double T = 0.25 * (grid[i+1][j] + grid[i-1][j] + grid[i][j-1] +
   grid[i][j+1]);
                double dT = fabs(T - grid[i][j]);
                new_grid[i][j] = T;
                if (dTmax < dT) dTmax = dT;
            }
        }

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

        if (dTmax_g < EPSILON) {
            finished = true;
        } else {
            for (size_t i = 0; i < N+2; i++) {
                upc_forall (size_t j = 0; j < N + 2; j++; &grid[i][j]) {
                    grid[i][j] = new_grid[i][j];
                }
            }
        }
        n_iter++;
        upc_barrier;
    } while (!finished);
    clock_t end = clock();

    if (MYTHREAD == 0) {
        double seconds = (double)(end - begin) / CLOCKS_PER_SEC;
        printf("%d iterations in %.3lf sec\n", n_iter, seconds);
        printf("Took %.3lf ms/iter\n", seconds * 1000.0 / n_iter);
    }
}
```

Listing 8: heat_1

2.3 Better memory use

One simple optimization is to avoid copying the destination array (`new_grid`) into the source array (`grid`) at the end of each iteration.

To do this, I implement a pointer flipping. Doing that decreases the amount of synchronization for each loop, which can be observed:

Threads	Time
2	$134.0\mu s$/iter
4	$103.9\mu s$/iter
8	$87.4\mu s$/iter
16	$81.4\mu s$/iter
32	$82.9\mu s$/iter

Table 6: Timing results for `heat_3.upc`

```
#include <upc_relaxed.h>
#include <bupc_collectivev.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdbool.h>

#define N 94
#define EPSILON 0.0001
#define TOTALSIZE ((N+2) * (N+2) / THREADS)

shared[TOTALSIZE] double grid[N+2][N+2];
shared[TOTALSIZE] double new_grid[N+2][N+2];
shared double dTmax[THREADS];

void init() {
    for (size_t j = 1; j < N+1; j++) {
        grid[0][j] = 1.0;
        new_grid[0][j] = 1.0;
    }
}

int main() {
    shared[TOTALSIZE] double (*ptr)[N+2] = grid;
    shared[TOTALSIZE] double (*new_ptr)[N+2] = new_grid;

    if (MYTHREAD == 0) {
        init();
    }

    upc_barrier;
    bool finished = false;
    int n_iter = 0;

    clock_t begin = clock();
    do {
        double dTmax = 0.0;

        if (n_iter % 2 == 0) {
            ptr = grid;
            new_ptr = new_grid;
        } else {
            ptr = new_grid;
            new_ptr = grid;
        }

        for (size_t i = 1; i <= N; i++) {
            upc_forall (size_t j = 1; j <= N; j++; &grid[i][j]) {
```

```
                double T = 0.25 * (ptr[i+1][j] + ptr[i-1][j] + ptr[i][j-1] + ptr[i][j+1]);
                double dT = fabs(T - ptr[i][j]);
                new_ptr[i][j] = T;
                if (dTmax < dT) dTmax = dT;
            }
        }

        // printf("%d: %lf\n", MYTHREAD, dTmax);

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

        if (dTmax_g < EPSILON) {
            finished = true;
        }
        n_iter++;
        // upc_barrier;
    } while (!finished);
    clock_t end = clock();

    if (MYTHREAD == 0) {
        double seconds = (double)(end - begin) / CLOCKS_PER_SEC;
        printf("%d iterations in %.3lf sec\n", n_iter, seconds);
        printf("Took %.3lf ms/iter\n", seconds * 1000.0 / n_iter);
    }
}
```

Listing 9: heat_3

2.4 Performance boost using privatization

As observed in the previous examples (Table 3), `upc_forall` has a higher performance cost than `for`.

To make use of less shared accesses, the program can copy a chunk of the *grid* array into private memory using `upc_memget` (into `ptr_priv` and `new_ptr_priv`).

I operate on the private array, only using remote accesses when necessary, and store the results in the new private array. Once the work is finished, the program put the new private array into `new_grid` with `upc_memput` and synchronize δ_{max}.

Threads	Time
2	$13.5\mu s$/iter
4	$12.6\mu s$/iter
8	$13.3\mu s$/iter
16	$15.0\mu s$/iter
32	$21.5\mu s$/iter

Table 7: Timing results for `heat_4.upc` (Listing 10)

```
#include <upc_relaxed.h>
#include <bupc_collectivev.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdbool.h>

```

```
#if ((N+2) % THREADS) != 0
#error N+2 must be divisible by THREADS
#endif

#define N 94
#define EPSILON 0.0001
// Change blocksize to (N+2)^2 / THREADS?
#define TOTALSIZE ((N+2) * (N+2) / THREADS)

#define LOCALWIDTH ((N+2) / THREADS)
#define LOCALSIZE (LOCALWIDTH * sizeof(double) * (N+2))

shared[TOTALSIZE] double grid[N+2][N+2];
shared[TOTALSIZE] double new_grid[N+2][N+2];
shared double dTmax[THREADS];

void init() {
    for (size_t j = 1; j < N+1; j++) {
        grid[0][j] = 1.0;
        new_grid[0][j] = 1.0;
    }
}

int main() {
    shared[TOTALSIZE] double (*ptr)[N+2] = grid;
    shared[TOTALSIZE] double (*new_ptr)[N+2] = new_grid;
    double (*ptr_priv)[N+2] = malloc(LOCALSIZE);
    double (*new_ptr_priv)[N+2] = malloc(LOCALSIZE);

    if (MYTHREAD == 0) {
        init();
    }

    upc_barrier;

    upc_memget(ptr_priv, &grid[LOCALWIDTH * MYTHREAD], LOCALSIZE);
    upc_memget(new_ptr_priv, &new_grid[LOCALWIDTH * MYTHREAD], LOCALSIZE);

    bool finished = false;
    int n_iter = 0;

    clock_t begin = clock();
    do {
        double dTmax = 0.0;

        size_t o = LOCALWIDTH * MYTHREAD;
        size_t i = 0;

        // Local block start
        if (i + o > 0) {
            for (size_t j = 1; j <= N; j++) {
                double T = 0.25 * (ptr_priv[i+1][j] + ptr[o+i-1][j] + ptr_priv[i][j-1] + ptr_priv[i][j+1]);
                // printf("%zu+%zu %zu: %lf\n", o, i, j, T);
                double dT = fabs(T - ptr_priv[i][j]);
                new_ptr_priv[i][j] = T;
                if (dTmax < dT) dTmax = dT;
            }
        }

        // Local block middle
```

```
        for (i += 1; i < LOCALWIDTH - 1; i++) {
            for (size_t j = 1; j <= N; j++) {
                double T = 0.25 * (ptr_priv[i+1][j] + ptr_priv[i-1][j] + ptr_priv[i][j-1] + ptr_priv[i][j+1]);
                // printf("%zu+%zu %zu: %lf\n", o, i, j, T);
                double dT = fabs(T - ptr_priv[i][j]);
                new_ptr_priv[i][j] = T;
                if (dTmax < dT) dTmax = dT;
            }
        }

        // Local block end
        if (i + o < N + 1) {
            for (size_t j = 1; j <= N; j++) {
                double T = 0.25 * (ptr[o+i+1][j] + ptr_priv[i-1][j] + ptr_priv[i][j-1] + ptr_priv[i][j+1]);
                // printf("%zu+%zu %zu: %lf\n", o, i, j, T);
                double dT = fabs(T - ptr_priv[i][j]);
                new_ptr_priv[i][j] = T;
                if (dTmax < dT) dTmax = dT;
            }
        }

        // printf("%d: %lf\n", MYTHREAD, dTmax);

        // upc_barrier;

        // Update ptr_priv and new_ptr_priv
        upc_memput(&new_ptr[LOCALWIDTH * MYTHREAD], new_ptr_priv, LOCALSIZE);

        // printf("%lf %lf\n", new_ptr_priv[1][1], new_ptr[o+1][1]);

        // Implicit barrier here:
        double dTmax_g = bupc_allv_reduce_all(double, dTmax, UPC_MAX);

        if (dTmax_g < EPSILON) {
            finished = true;
        } else {
            // Swap ptr and new_ptr
            shared[TOTALSIZE] double (*ptr_tmp)[N+2] = ptr;
            ptr = new_ptr;
            new_ptr = ptr_tmp;

            double (*ptr_tmp_priv)[N+2] = ptr_priv;
            ptr_priv = new_ptr_priv;
            new_ptr_priv = ptr_tmp_priv;
        }
        n_iter++;
        // upc_barrier;
    } while (!finished);
    clock_t end = clock();

    if (MYTHREAD == 0) {
        double seconds = (double)(end - begin) / CLOCKS_PER_SEC;
        printf("%d iterations in %.3lf sec\n", n_iter, seconds);
        printf("Took %.3lf ms/iter\n", seconds * 1000.0 / n_iter);
    }
}
```

Listing 10: heat_4.upc

2.5 Dynamic problem size

This last step is to make the problem size dynamic, in other words specifying N at runtime. This is made possible by the use of dynamic allocating shared memory during execution.

I had to adapt the code to give N as a command line parameter.

Because speed for this part is not a worry, I based the code from `heat_3.upc`, as the previous optimization step made the code harder to work with.

The declaration of `grid` and `new_grid` are now done through the function `upc_all_alloc`. The matrix notation, while useful, had to be deleted, as it was dependent on N, which is now dynamic. This meant re-writing all the matrix accesses to now multiply the y coordinate by $(n + 2)$ and to add its result to the x coordinate.

```c
#include <upc_relaxed.h>
#include <upc_collective.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdbool.h>
#include <stdlib.h>

#define N 94
#define EPSILON 0.0001
shared double dTmax_g;

void init(shared[] double* grid, shared[] double* new_grid, size_t n) {
    for (size_t i = 0; i < n+2; i++) {
        for (size_t j = 0; j < n+2; j++) {
            grid[i * (n+2) + j] = 0.0;
            new_grid[i * (n+2) + j] = 0.0;
        }
    }

    for (size_t j = 1; j < n+1; j++) {
        grid[j] = 1.0;
        new_grid[j] = 1.0;
    }
}

int main(int argc, char* argv[]) {
    if (argc != 2) {
        if (MYTHREAD == 0) fprintf(stderr, "Error: expected two arguments, got %d\n", argc);
        exit(1);
    }

    size_t 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));

    // Handy way to access ptr from now on
    #define ptr_get(x, y) ptr[(x) * (n+2) + (y)]

    if (MYTHREAD == 0) {
        init(ptr, new_ptr, n);
```

```
    }

    upc_barrier;
    bool finished = false;
    int n_iter = 0;

    clock_t begin = clock();
    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 (size_t j = 1; j <= n; j++) {
                double T = 0.25 * (ptr_get(i+1, j) + ptr_get(i-1, j) + ptr_get(i,
   j-1) + ptr_get(i, j+1));
                double 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 = true;
        } else {
            shared[] double* tmp = ptr;
            ptr = new_ptr;
            new_ptr = tmp;
        }
        n_iter++;
        // upc_barrier;
    } while (!finished);
    clock_t end = clock();

    if (MYTHREAD == 0) {
        double seconds = (double)(end - begin) / CLOCKS_PER_SEC;
        printf("%d iterations in %.3lf sec\n", n_iter, seconds);
        printf("Took %.3lf ms/iter\n", seconds * 1000.0 / n_iter);
    }
}
```

Listing 11: `heat_5.upc`

3 Conclusion

The simplified 1D Laplace Equation Solver (Section 1) was really beginner-friendly and guided in order to learn how to use the UPC. Contrary to the 2D Heat Algorithm (Section 2), which was a little more difficult but gave a realistic and real example to see what the UPC can do in real situations.

Unfortunately, the overhead encountered here is mainly caused by the shared pointer operations. Minimizing this overhead required heavy modifications. The UPC implementation required at least 8 threads to overcome the language's overhead.

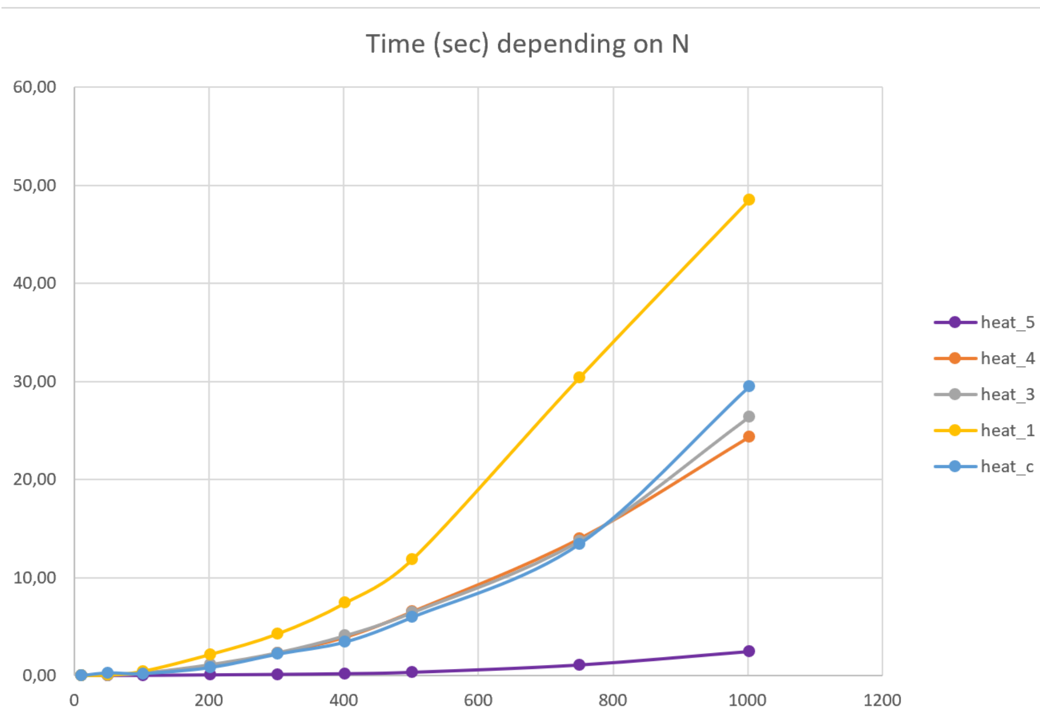
This difficulty to beat the speed of C is explained by the fact that each iteration consists of a small amount of work over a large set of data, that needs to be synchronized before the next iteration. Even with a perfect parallelization of the algorithm, the slowest thread contributing to the calculation, this will be restrictive.

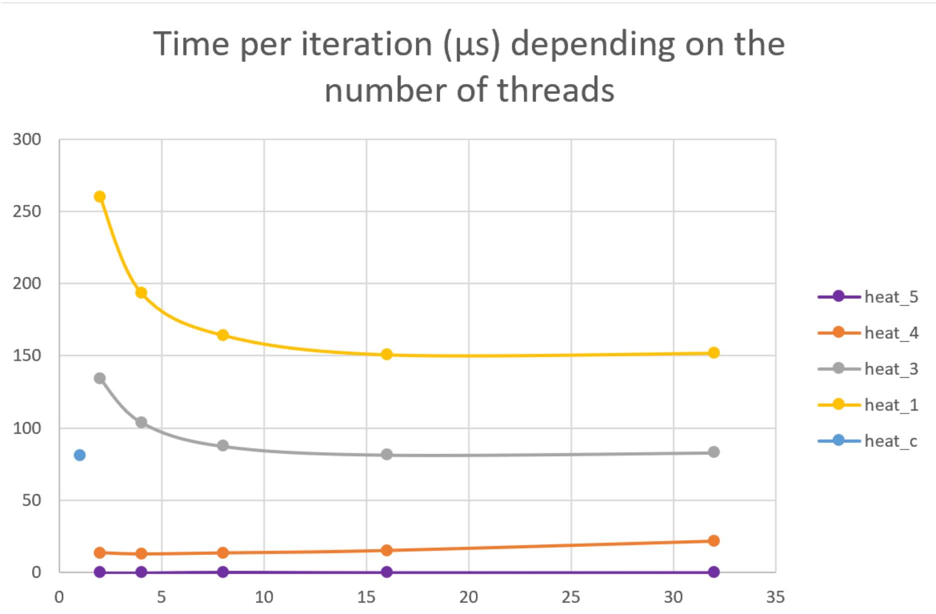
3.1 Speed comparison

The Simplified 1D Laplace Equation Solver scored great until I introduced δ_{max}, which required some costly synchronization.

For the 2D Heat, the synchronization costs and the mesoshared server made it unfeasible to use a greater amount of threads.

Following are two comparative graphics of all the measurements of the 2D Heat's section.




Time per iteration (µs) depending on the number of threads
300
250
200
150
100
50
0
0
5
10
15
20
25
30
35
heat_5
heat_4
heat_3
heat_1
heat_c