PC40 Hands-on: UPC

Adrien Burgun

Automne 2021

Abstract

For this hands-on assignement, I wanted to measure the performance of the different versions of the code and compare them to measure speedups. For this reason, my code diverges slightly from the template that was given to us, as I needed to insert a timing method and avoid refactoring the code.

You will find in this report listings of the different versions of the code, alongside experimental measurements and commentaries.

The source code itself, this report's source code and instructions on how to build and run the code yourself can be found on this project's git repository: `https://github.com/adri326/pc40-upc/`.

Contents

1 Simplified 1D Laplace solver

1.1 C implementation

The C implementation of the laplace solver has been slightly modified to time the main loop. It is single-threaded but the implementation allows for compiler SIMD optimizations. Settings common to every version (vector size, epsilon, max number of iterations) have been placed in a file called `settings.h`.

This code has been compiled with `gcc v4.9.0` and run on the `mesoshared` server, yielding the following results:

Options	Time (avg)	CI ($\sigma = 0.01$)
`-O0`	$450\mu s/\text{iter}$	$\pm\ 10\mu s/\text{iter}$
`-O3`	$140\mu s/\text{iter}$	$\pm\ 10\mu s/\text{iter}$

Table 1: Timing results for the C implementation of the 1D Laplace solver (Listing 1)

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

#define LEN DESIRED_LEN

void init();
void iteration();
void copy_array();

double x_new[LEN];
double x[LEN];
double b[LEN];

int main() {
    init();

    clock_t begin = clock();
    for (size_t n = 0; n < ITERATIONS; n++) {
        iteration();
        if (n != ITERATIONS - 1) copy_array();
    }
    clock_t end = clock();

    double diff = (double)(end - begin) / CLOCKS_PER_SEC / ITERATIONS;
    double mean_squared = 0.0;
    if (DISPLAY) {
        printf("| b      | x      | x_new  | diff    |\n");
        printf("| :----- | :----- | :----- | :------ |\n");
        for (size_t i = 0; i < LEN; i++) {
            printf("| %1.4lf | %1.4lf | %1.4lf | % 1.4lf |\n", b[i], x[i], x_new[i], x_new[i] - x[i]);
        }
    }
    for (size_t i = 0; i < LEN; i++) mean_squared += (x_new[i] - x[i]) * (x_new[i] - x[i]);
    printf("\nIterations: %d\n", ITERATIONS);
    printf("Mean squared difference: %1.4lf\n", mean_squared / LEN);
    printf("Took %.3lfms/iter!\n", diff * 1000);
}

void init() {
    srand(time(0));
    for (size_t i = 0; i < LEN; i++) {
        b[i] = (double)rand() / RAND_MAX;
        x[i] = (double)rand() / RAND_MAX;
        x_new[i] = 0;
    }
}

void iteration() {
    for (size_t i = 1; i < LEN - 1; i++) {
        x_new[i] = 0.5 * (x[i-1] + x[i+1] + b[i]);
    }
    x_new[0] = x[0];
    x_new[LEN - 1] = x[LEN - 1];
}

void copy_array() {
    for (size_t i = 0; i < LEN; i++) {
        x[i] = x_new[i];
    }
}
```

Listing 1: C implementation of the 1D Laplace solver

1.2 Porting the C code to UPC

The base C implementation 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, so additional conditionals were added when needed.

Finally, the `x`, `xnew` and `b` arrays were made shared and `upc_barrier` statements were added at the end of `init`, `iteration` and `copy_array`: this is 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 early and printing the wrong timing.

This code was compiled with `upcc v2.22.0 + gcc v4.2.4` and run on the `mesoshared` server, yielding the following results:

Threads	Time (avg)	CI ($\sigma = 0.01$)
2	$515.2\mu s/\text{iter}$	$\pm\ 3.9\mu s/\text{iter}$
3	$536.2\mu s/\text{iter}$	$\pm\ 2.4\mu s/\text{iter}$
4	$312.4\mu s/\text{iter}$	$\pm\ 2.0\mu s/\text{iter}$
8	$204.0\mu s/\text{iter}$	$\pm\ 2.0\mu s/\text{iter}$
16	$166.2\mu s/\text{iter}$	$\pm\ 2.5\mu s/\text{iter}$
32	$150.4\mu s/\text{iter}$	$\pm\ 3.2\mu s/\text{iter}$

Table 2: Timing results for the first UPC implementation of the 1D Laplace solver (Listing 2)

When compiled and run with 3 threads, the code runs noticeably slower. For curiosity, I ran the code with 24 and 31 threads and obtained a similar slowdown:

$$\text{Threads} = 24, \quad \text{Time} = 252.6\mu s/\text{iter} \pm 3.7\mu s/\text{iter} \quad (\text{expected} \approx 160\ \mu s/\text{iter})$$

Threads = 31, Time = $317.6\mu s/\text{iter} \pm 4.4\mu s/\text{iter}$ (expected $\approx 150\ \mu s/\text{iter}$)

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

#define LEN DESIRED_LEN

void init();
void iteration();
void copy_array();

shared double x_new[LEN];
shared double x[LEN];
shared double b[LEN];

int main() {
    init();

    clock_t begin = clock();
    for (size_t n = 0; n < ITERATIONS; n++) {
        iteration();
        if (n != ITERATIONS - 1) copy_array();
    }
    clock_t end = clock();

    if (MYTHREAD == 0) {
        double diff = (double)(end - begin) / CLOCKS_PER_SEC / ITERATIONS;
        double mean_squared = 0.0;
        if (DISPLAY) {
            printf("| b        | x        | x_new    | diff      |\n");
            printf("| :------- | :------- | :------- | :-------- |\n");
            for (size_t i = 0; i < LEN; i++) {
                printf("| %1.4lf | %1.4lf | %1.4lf | % 1.4lf |\n", b[i], x[i], x_new[i], x_new[i] - x[i]);
            }
        }
        for (size_t i = 0; i < LEN; i++) mean_squared += (x_new[i] - x[i]) * (x_new[i] - x[i]);
        printf("\nIterations: %d\n", ITERATIONS);
        printf("Mean squared difference: %1.4lf\n", mean_squared / LEN);
        printf("Took %.3lfms/iter!\n", diff * 1000);
    }
}

void init() {
    if (MYTHREAD == 0) {
        srand(time(0));
        for (size_t i = 0; i < LEN; i++) {
            b[i] = (double)rand() / RAND_MAX;
            x[i] = (double)rand() / RAND_MAX;
            x_new[i] = 0;
        }
    }
    upc_barrier;
}

void iteration() {
    for (size_t i = 1; i < LEN - 1; i++) {
        if (i % THREADS == MYTHREAD) {
            x_new[i] = 0.5 * (x[i-1] + x[i+1] + b[i]);
        }
    }

    if (MYTHREAD == 0) x_new[0] = x[0];
    if (MYTHREAD == (LEN - 1) % THREADS) x_new[LEN - 1] = x[LEN - 1];

    upc_barrier;
}

void copy_array() {
    for (size_t i = 0; i < LEN; i++) {
        if (i % THREADS == MYTHREAD) x[i] = x_new[i];
    }
    upc_barrier;
}
```

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

1.3 Optimizing the inner for loop

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

To achieve this, only the `iteration()` function had to be changed. This transformation is shown in Figure 1

This change greatly increases the speed of the program:

Threads	Time (avg)	CI ($\sigma = 0.01$)
2	$482.0\mu s/\text{iter}$	$\pm\ 3.2\mu s/\text{iter}$
4	$265.2\mu s/\text{iter}$	$\pm\ 2.3\mu s/\text{iter}$
8	$128.8\mu s/\text{iter}$	$\pm\ 1.6\mu s/\text{iter}$
16	$74.6\mu s/\text{iter}$	$\pm\ 3.1\mu s/\text{iter}$
32	$46.0\mu s/\text{iter}$	$\pm\ 2.5\mu s/\text{iter}$

Table 3: Timing results for the second UPC implementation of the 1D Laplace solver

1.4 Optimizing the array blocking factor

Our next optimization is to increase the blocking factor B 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 of the current thread's affinity. With a greater block factor, these outside accesses can be reduced to only $2/B$ accesses on average.

Two parts of the code had to be changed: the array declarations (Figure 2) and the loop in `iteration()` and `copy_array()` (Figure 3).

The effects of this change is dependent on B: similar to the results found in Table 2, the code runs fastest when $B = 2^n$. Whichever value is chosen, however, the program did not run faster

```
// ex3.upc:
void iteration() {
  for (size_t i = 1; i < LEN - 1; i++) {
      if (i % THREADS == MYTHREAD) {
          x_new[i] = 0.5 * (x[i-1] + x[i
   +1] + b[i]);
      }
  }

  if (MYTHREAD == 0) x_new[0] = x[0];
  if (MYTHREAD == (LEN - 1) % THREADS)
   x_new[LEN - 1] = x[LEN - 1];

  upc_barrier;
}
```

(a) Previous `iteration()` function from Listing 2

```
// ex4.upc:
void iteration() {
    size_t i = MYTHREAD;
    if (MYTHREAD == 0) {
        i += THREADS;
        x_new[0] = x[0];
    }

    for (; i < LEN - 1; i += THREADS) {
        x_new[i] = 0.5 * (x[i-1] + x[i
   +1] + b[i]);
    }

    if (MYTHREAD == THREADS - 1) x_new[
   LEN - 1] = x[LEN - 1];

    upc_barrier;
}
```

(b) New `iteration()` function without the inner `if`

Figure 1: Modification in `iteration()` to remove the reduce the number of loop iterations.

than `ex4.upc` (Table 3), but rather ran much slower (the source code of this version can be found here):

Threads	Time (avg)	CI ($\sigma = 0.01$)
2	$959.2\mu s/\text{iter}$	$\pm\ 3.9\mu s/\text{iter}$
4	$863.9\mu s/\text{iter}$	$\pm\ 16.1\mu s/\text{iter}$
8	$691.6\mu s/\text{iter}$	$\pm\ 6.1\mu s/\text{iter}$
16	$581.5\mu s/\text{iter}$	$\pm\ 4.1\mu s/\text{iter}$
32	$536.9\mu s/\text{iter}$	$\pm\ 4.0\mu s/\text{iter}$

Table 4: Timing results for the third UPC implementation of the 1D Laplace solver, $B = 32$

When ran with $B = 1$, $THREADS = 32$, the code takes $318\mu s/\text{iter} \pm 7.8\mu s/\text{iter}$. I do not know what causes this slowdown, but it seems like `upc_forall` could contribute to it.

When limiting the modification of `ex4.upc` to only replace the `for` loop with a `upc_forall` loop, the example becomes 83% slower ($206\mu s/\text{iter} \rightarrow 376\mu s/\text{iter}$, 8 threads, AMD Ryzen 2700X, `upcc v2.28.2 + gcc v9.3.0`).

1.4.1 Note on the update/copy loop

So far, the program has been running many iterations of the "main" loop, which calls `iteration()` and `copy_array()`. A `upc_barrier;` call is necessary at the end of each of those two operations, as `copy_array()` modifies x and depends on the processed value of x_{new}, while `iteration()` modifies x_{new} and depends on the copied value in x.

```
// ex5.upc:
void iteration() {
    upc_forall (size_t i = 1; i < LEN - 1; i++; &x_new[i]) {
        x_new[i] = 0.5 * (x[i-1] + x[i+1] + b[i]);
    }
```

```
// ex4.upc:
shared double x_new[LEN];
shared double x[LEN];
shared double b[LEN];
```

(a) Previous array declarations

```
// settings.h:
#define BLOCKSIZE 32

// ex5.upc:
shared[BLOCKSIZE] double x_new[LEN];
shared[BLOCKSIZE] double x[LEN];
shared[BLOCKSIZE] double b[LEN];
```

(b) New array declarations with blocking factor

Figure 2: Modification of the array declarations to increase the blocking factor B.

```
    if (MYTHREAD == 0) x_new[0] = x[0];
    if (MYTHREAD == THREADS - 1) x_new[LEN - 1] = x[LEN - 1];

    upc_barrier;
}

void copy_array() {
    upc_forall (size_t i = 0; i < LEN; i++; &x_new[i]) {
        x[i] = x_new[i];
    }
    upc_barrier;
}
```

Listing 3: New `iteration()` and `copy_array()` implementations

1.5 Detecting convergence

One of our 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 $\delta_{max} \leqslant \varepsilon$.

Because of the shared nature of x and x_{new}, we need to compute a partial δ^t_{max} and compute $\delta_{max} = \max_{t\in[0;T[}(\delta^t_{max})$, with:

$$\delta^t_{max} = \max_{i\in[\![0;n-1]\!],\ \text{affinity}(x_{new}[i])=t} \Big| x[i] - x_{new}[i] \Big|$$

The new timings are:

Threads	Time (avg)	CI ($\sigma = 0.01$)
2	$1400.4\mu s/$iter	$\pm$ $6.0\mu s/$iter
4	$1092.8\mu s/$iter	$\pm$ $17.0\mu s/$iter
8	$803.4\mu s/$iter	$\pm$ $7.7\mu s/$iter
16	$647.8\mu s/$iter	$\pm$ $18.9\mu s/$iter
32	$572.0\mu s/$iter	$\pm$ $4.6\mu s/$iter

Table 5: Timing results for the fourth UPC implementation of the 1D Laplace solver, $B = 32$, `MAX_ITER = 1000`

```
// ex6.upc (partial):
shared double diff[THREADS];
shared double diffmax;

void handle_diff(double d) {
    if (d < 0.0) d = -d;
    if (diff[MYTHREAD] < d) diff[MYTHREAD] = d;
}

int main() {
    // ...
    while (true) {
        iteration();

        upc_barrier;

        if (MYTHREAD == 0) {
            for (size_t n = 0; n < THREADS; n++) {
                if (diffmax < diff[n]) diffmax = diff[n];
            }
        }

        upc_barrier;

        if (diffmax <= EPSILON) break;
        if (++iter > MAX_ITER) break;

        copy_array(); // upc_barrier; at the end of copy_array();
    }
    // ...
}

void iteration() {
    upc_forall (size_t i = 1; i < LEN - 1; i++; &x_new[i]) {
        x_new[i] = 0.5 * (x[i-1] + x[i+1] + b[i]);
        handle_diff(x_new[i] - x[i]);
    }
}
```
Listing 4: Excerpt from ex6.upc: new `handle_diff` function and calculation of `diffmax`

1.5.1 Using reduction operations

We can further optimize this code by using reduction operations from `upc_collective.h`. The optimized variant is available at `laplace/optimized.upc`.

By removing the calls to `upc_forall`, we can reduce the time taken by a further 35%($\pm 5\% pt$).

Threads	Time (avg)	CI ($\sigma = 0.01$)
2	1216.0μs/iter	$\pm$ 27.1μs/iter
4	778.0μs/iter	$\pm$ 15.1μs/iter
8	415.8μs/iter	$\pm$ 10μs/iter
16	225.8μs/iter	$\pm$ 4.8μs/iter
32	130.0μs/iter	$\pm$ 3.1μs/iter
64	98.2μs/iter	$\pm$ 2.7μs/iter

Table 6: Timing results for the fully optimized UPC implementation of the 1D Laplace solver, $B = 32$, `MAX_ITER = 1000`

1.6 Conclusion (1D Laplace Solver)

The first implementation in UPC of the 1D Laplace solver algorithm and its subsequent first optimization (flattening the `for (...) if (...)`) gave very promising results for the improvement of the speed of the program, bringing the timings down from 140μs/iteration to only 46μs/iteration.

Unfortunately, the next optimization attempt, which introduces a blocking factor to the primary arrays and used `upc_forall`, brought the speeds down and made the UPC implementation much slower than the C implementation.

The addition of δ_{max}, as a way to stop the algorithm once a sufficiently accurate solution is found, inevitably added another overhead to the program. With my best efforts, I could only bring its performance to somewhere between those of the first implementation and its first optimization.

Nonetheless, the resulting program runs slightly faster than the original C implementation, given enough threads).

2 2D Heat conduction

For this second algorithm, we will try to improve on the knowledge gathered from the first algorithm and implement a simple, 2D heat conduction simulation. As with Section 1, we begin with a simple C implementation of the algorithm, which will work as our performance base value.

The provided template came bundled with a performance measurement method, but I swapped it out with `clock()` in the subsequent UPC implementations for consistency with the last section's code.

2.1 First C implementation

Following are the performance results for the first C implementation, ran on `mesoshared` with `gcc v4.9.0` and on an `AMD Ryzen 2700X` with `clang v12.0.1`:

Options	Time (avg)	CI ($\sigma = 0.01$)
`-O0`	$89.1\mu s/\text{iter}$	$\pm\ 0.3\mu s/\text{iter}$
`-O3`	$15.0\mu s/\text{iter}$	$\pm\ 0.1\mu s/\text{iter}$

Table 7: Timing results for `heat_c.c` (mesoshared, Listing 5)

Options	Time (avg)	CI ($\sigma = 0.01$)
`-O0`	$80.9\mu s/\text{iter}$	$\pm\ 2.6\mu s/\text{iter}$
`-O3`	$17.9\mu s/\text{iter}$	$\pm\ 2.2\mu s/\text{iter}$

Table 8: Timing results for `heat_c.c` (Ryzen 2700X, Listing 5)

```c
#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include <stdbool.h>
#include "settings.h"

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_start, ts_end;
    double dTmax, dT, time;
    int i, j, k, l;
    bool finished = false;
    double T;
    int n_iter = 0;

    initialize();

    /* Set the precision wanted */
    finished = 0;

    /* and start the timed section */
    gettimeofday(&ts_start, 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 = true;
        } else { // It isn't: preparing for a new iteration
            for (k=0; k<N+2; k++) {
                for (l=0; l<N+2; l++) {
                    grid[k][l] = new_grid[k][l];
                }
            }
        }
        n_iter++;
    } while (!finished);

    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_start.tv_sec + (ts_start.tv_usec / 1000000.0);

    printf("%d iterations in %.3lf sec\n", n_iter, time);
    printf("Took %.3lf ms/iter\n", time * 1000.0 / n_iter);

    return 0;
}
```

Listing 5: C implementation of the 2D Heat simulation

2.2 Porting the C code to UPC

Porting the template C version to UPC is straightforward. The performance, however, takes a hit, as the default work sharing causes a lot of remote accesses to memory, despite the blocking factor that was put in place. We obtain the following measurements on `mesoshared`:

Threads	Time (avg)	CI ($\sigma = 0.01$)
2	$259.9\mu s/\text{iter}$	$\pm\ 1.9\mu s/\text{iter}$
4	$193.6\mu s/\text{iter}$	$\pm\ 1.5\mu s/\text{iter}$
8	$164.1\mu s/\text{iter}$	$\pm\ 1.2\mu s/\text{iter}$
16	$150.6\mu s/\text{iter}$	$\pm\ 1.4\mu s/\text{iter}$
32	$151.7\mu s/\text{iter}$	$\pm\ 1.7\mu s/\text{iter}$

Table 9: Timing results for `heat_1.upc`

```
// heat_c.c:
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 */
  }
}
```

(a) Main loop in `heat_c.c`

```
// heat_1.upc:
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;
  }
}
```

(b) Main loop in `heat_1.upc`

Figure 3: Porting the main loop from C to UPC

2.3 Optimizing the array accesses

One simple optimization is to avoid copying the destination array (`new_grid`) into the source array (`grid`).

To do this, we store two shared pointers that will point on either array, and we swap them at the end of each iteration, simulating a copy of `grid` into `new_grid`, with a $\mathcal{O}(1)$ time complexity instead of $\mathcal{O}(n)$. Doing so effectively halves the amount of synchronization that needs to happen for each loop, which can be observed in the timing results:

Threads	Time (avg)	CI ($\sigma = 0.01$)
2	$134.0\mu s$/iter	$\pm$ $1.4\mu s$/iter
4	$103.9\mu s$/iter	$\pm$ $4.1\mu s$/iter
8	$87.4\mu s$/iter	$\pm$ $1.0\mu s$/iter
16	$81.4\mu s$/iter	$\pm$ $1.2\mu s$/iter
32	$82.9\mu s$/iter	$\pm$ $1.5\mu s$/iter

Table 10: Timing results for `heat_3.upc`

This optimization was also done on the C version for comparison, and we obtain the following speeds:

Options	Time (avg)	CI ($\sigma = 0.01$)
`-O0`	$64.0\mu s$/iter	$\pm$ $0.5\mu s$/iter
`-O3`	$11.3\mu s$/iter	$\pm$ $0.4\mu s$/iter

Table 11: Timing results for `heat_c_ptr.c`

```
// heat_3.upc:
shared[BLOCKSIZE] double (*ptr)[N+2] = grid;
shared[BLOCKSIZE] double (*new_ptr)[N+2] = new_grid;

// ...

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

// ...

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;
```

Listing 6: Excerpt from `heat_3.upc`, where pointer-swapping was implemented

2.4 Array privatization

As observed in the later examples of the 1D Laplace equation solver (Table 4), `upc_forall` has a significant performance cost over a regular `for` loop. We also know that local shared memory accesses have a considerable cost when the shared pointer math is done without dedicated hardware support [1].

To make use of less shared accesses, we can copy a chunk of the *grid* array into private memory using `upc_memget` (into `ptr_priv` and `new_ptr_priv`). We specifically only retrieve the memory that has the affinity to the current thread, so that we can write to it and broadcast it in one go. The memory is copied also duplicated to make use of pointer-swapping as in Section 2.3.

We then 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, we put the new private array into `new_grid` with `upc_memput` and synchronize δ_{max}.

Threads	Time (avg)	CI ($\sigma = 0.01$)
2	$13.5\mu s$/iter	$\pm$ $1.4\mu s$/iter
4	$12.6\mu s$/iter	$\pm$ $1.2\mu s$/iter
8	$13.3\mu s$/iter	$\pm$ $1.0\mu s$/iter
16	$15.0\mu s$/iter	$\pm$ $1.2\mu s$/iter
32	$21.5\mu s$/iter	$\pm$ $1.1\mu s$/iter

Table 12: Timing results for `heat_4.upc` (Listing 7)

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

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

// Change blocksize to (N+2)^2 / THREADS?
#define BLOCKSIZE ((N+2) * (N+2) / THREADS)
// #define BLOCKSIZE ((N+2) * THREADS)
#define LOCALWIDTH ((N+2) / THREADS)
#define LOCALSIZE (LOCALWIDTH * sizeof(double) * (N+2))

shared[BLOCKSIZE] double grid[N+2][N+2];
shared[BLOCKSIZE] 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[BLOCKSIZE] double (*ptr)[N+2] = grid;
    shared[BLOCKSIZE] 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[BLOCKSIZE] 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 7: Optimized UPC implementation of the 2D Heat algorithm (`heat_4.upc`)

2.5 Dynamic problem size

This last version of our code is not an optimization: we adapt it so that we can specify the number of threads and the size of the grid at runtime, rather than at compile time. This requires changing a big portion of the code, as it was previously using pre-defined constants for these two parameters.

Because speed for this part is not a worry, we 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` have been deleted and are done implicitely through the compiler-supplied function `upc_all_alloc`. The matrix notation, while useful, had to be gotten rid of, as it was dependent on N, which is now dynamic. This meant re-writing all of the matrix accesses to now multiply the y coordinate by $(n + 2)$ and to add its result to the x coordinate.

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

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++) {
```

```c
            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 8: Dynamic UPC implementation of the 2D Heat algorithm (`heat_5.upc`)

3 Conclusion

For both the Simplified 1D Laplace Equation Solver (Section 1) and the 2D Heat Algorithm (Section 2), we were able to take a simple C implementation and port it to UPC with minimal effort. These two problems are easily parallelizable, as all of the work within an iteration can be done independently (or vertically).

We then went along the process of refining the UPC implementation and optimizing the various accesses to the arrays to reduce the overhead of parallelization induced by UPC. Unfortunately, the overhead encountered here is mainly caused by the shared pointer operations (as highlighted by the results in Section 1.4 and Section 2.4). Minimizing this overhead required heavy modification to the code.

With all of our efforts, the UPC implementation only came shortly before the C implementation when benchmarked and required at least 8 threads to overcome the language's overhead. This difficulty to beat the speed of C is explained by the nature of the problems at hand: 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 (with no synchronization overhead), we would still be bottlenecked by the data transfer speeds and the slowest thread contributing to the calculation.

3.1 Speed comparison

Following are two comparative tables of all of the measurements of this report.

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

The 2D Heat Algorithm did worse, only achieving speeds close to those of C. The synchronization costs made it unfeasible to use a greater amount of threads, as can be seen with the last few rows.

Implementation	Options	Time (avg)	CI ($\sigma = 0.01$)
`laplace/ex2.c`	`-O0`	$450\mu s/\text{iter}$	$\pm\ 10\mu s/\text{iter}$
	`-O3`	$140\mu s/\text{iter}$	$\pm\ 10\mu s/\text{iter}$
`laplace/ex3.upc` (w/o δ_{max})	$T = 2$	$515.2\mu s/\text{iter}$	$\pm\ 3.9\mu s/\text{iter}$
	$T = 4$	$312.4\mu s/\text{iter}$	$\pm\ 2.0\mu s/\text{iter}$
	$T = 8$	$204.0\mu s/\text{iter}$	$\pm\ 2.0\mu s/\text{iter}$
	$T = 16$	$166.2\mu s/\text{iter}$	$\pm\ 2.5\mu s/\text{iter}$
	$T = 32$	$150.4\mu s/\text{iter}$	$\pm\ 3.2\mu s/\text{iter}$
`laplace/ex4.upc` (w/o δ_{max})	$T = 2$	$482.0\mu s/\text{iter}$	$\pm\ 3.2\mu s/\text{iter}$
	$T = 4$	$265.2\mu s/\text{iter}$	$\pm\ 2.3\mu s/\text{iter}$
	$T = 8$	$128.8\mu s/\text{iter}$	$\pm\ 1.6\mu s/\text{iter}$
	$T = 16$	$74.6\mu s/\text{iter}$	$\pm\ 3.1\mu s/\text{iter}$
	$T = 32$	$46.0\mu s/\text{iter}$	$\pm\ 2.5\mu s/\text{iter}$
`laplace/ex5.upc` (with δ_{max})	$T = 2$	$959.2\mu s/\text{iter}$	$\pm\ 3.9\mu s/\text{iter}$
	$T = 4$	$863.9\mu s/\text{iter}$	$\pm\ 16.1\mu s/\text{iter}$
	$T = 8$	$691.6\mu s/\text{iter}$	$\pm\ 6.1\mu s/\text{iter}$
	$T = 16$	$581.5\mu s/\text{iter}$	$\pm\ 4.1\mu s/\text{iter}$
	$T = 32$	$536.9\mu s/\text{iter}$	$\pm\ 4.0\mu s/\text{iter}$
`laplace/ex6.upc` (with δ_{max})	$T = 2$	$1400.4\mu s/\text{iter}$	$\pm\ 6.0\mu s/\text{iter}$
	$T = 4$	$1092.8\mu s/\text{iter}$	$\pm\ 17.0\mu s/\text{iter}$
	$T = 8$	$803.4\mu s/\text{iter}$	$\pm\ 7.7\mu s/\text{iter}$
	$T = 16$	$647.8\mu s/\text{iter}$	$\pm\ 18.9\mu s/\text{iter}$
	$T = 32$	$572.0\mu s/\text{iter}$	$\pm\ 4.6\mu s/\text{iter}$
`laplace/optimized.upc` (with δ_{max})	$T = 2$	$1216.0\mu s/\text{iter}$	$\pm\ 27.1\mu s/\text{iter}$
	$T = 4$	$778.0\mu s/\text{iter}$	$\pm\ 15.1\mu s/\text{iter}$
	$T = 8$	$415.8\mu s/\text{iter}$	$\pm\ 10\mu s/\text{iter}$
	$T = 16$	$225.8\mu s/\text{iter}$	$\pm\ 4.8\mu s/\text{iter}$
	$T = 32$	$130.0\mu s/\text{iter}$	$\pm\ 3.1\mu s/\text{iter}$
	$T = 64$	$98.2\mu s/\text{iter}$	$\pm\ 2.7\mu s/\text{iter}$

Table 13: Comparative table of the benchmarks of the Simplified Laplace 1D Solver

Implementation	Options	Time (avg)	CI ($\sigma = 0.01$)
heat_c.c (w/o ptr-swap)	-O0	80.9μs/iter	$\pm$ 2.6μs/iter
	-O3	17.9μs/iter	$\pm$ 2.2μs/iter
heat_c_ptr.c (with ptr-swap)	-O0	64.0μs/iter	$\pm$ 0.5μs/iter
	-O3	11.3μs/iter	$\pm$ 0.4μs/iter
heat_1.upc (w/o ptr-swap)	$T = 2$	259.9μs/iter	$\pm$ 1.9μs/iter
	$T = 4$	193.6μs/iter	$\pm$ 1.5μs/iter
	$T = 8$	164.1μs/iter	$\pm$ 1.2μs/iter
	$T = 16$	150.6μs/iter	$\pm$ 1.4μs/iter
	$T = 32$	151.7μs/iter	$\pm$ 1.7μs/iter
heat_3.upc (with ptr-swap)	$T = 2$	134.0μs/iter	$\pm$ 1.4μs/iter
	$T = 4$	103.9μs/iter	$\pm$ 4.1μs/iter
	$T = 8$	87.4μs/iter	$\pm$ 1.0μs/iter
	$T = 16$	81.4μs/iter	$\pm$ 1.2μs/iter
	$T = 32$	82.9μs/iter	$\pm$ 1.5μs/iter
heat_4.upc (with ptr-swap)	$T = 2$	13.5μs/iter	$\pm$ 1.4μs/iter
	$T = 4$	12.6μs/iter	$\pm$ 1.2μs/iter
	$T = 8$	13.3μs/iter	$\pm$ 1.0μs/iter
	$T = 16$	15.0μs/iter	$\pm$ 1.2μs/iter
	$T = 32$	21.5μs/iter	$\pm$ 1.1μs/iter

Table 14: Comparative table of the benchmarks of the 2D Heat Algorithm

3.2 Personal remarks

Overall, I enjoyed working with UPC and optimizing the parallelized code. However, I feel like UPC is lacking behind in terms of what the compiler can optimize for the developer. For instance, it could unwrap a `upc_forall` loop into two `for` loops when the increment is a divisor of the blocking factor: this can easily be done with macros already and can only make the code faster.

The hardest part of working on this project was finding documentation for the language itself. The documentation is scattered across internet, with some tutorials explaining how basic functions work but missing the more complicated functions, some documentation hosted on code mirroring websites and the rest burried inside of the compiler's source code.

Searching for example *"upc_all_reduceD"* online brings up an 11-year old mail archive, a benchmark of UPC and quite further down the summary of the UPC Collective specification. The only way to truly know what this function does is to read its source code.

This contrasts with other, smaller languages that I have encountered until now, whose documentation is usually centered in one place. With the similarly niche language "Pony", searching *"ponylang hashmap"* online yields a rendered version of the documentation of the language's standard library as the first result.

The errors outputted by the language were also of little help, as they did not show any additional information. The `-verbose` flag only displayed information unrelated to the error.

Nonetheless, I am glad that I got to try out this language and work with parallelized code.

```
$ upcc 2d_heat/heat_5.upc -o build/heat_5
upcc: error during UPC-to-C translation (sgiupc stage):
2d_heat/heat_5.upc: In function 'main':
2d_heat/heat_5.upc:66: incompatible type for argument 1 of 'bupc_all_reduceD'

```

Figure 4: *"I guess the compiler wants me to learn the function signature by myself."*

References

[1] Olivier Serres. *Hardware Support for Productive Partitioned Global Address Space (PGAS) Programming*. PhD thesis, George Washington University, 2015.