PC40 Report

UV Professor J.Gaber

Fall 2022

Raphaël PERRIN

October 30, 2022

Table of Contents

1 Introduction

The aim of this TP was to put into practice the concept of parallel computing using Unified Parallel C (UPC), an extension of the programming language C. Thus, we learned the using of shared variables, tables and pointers that can be directly read and written by any thread of our program.

To do so, during our Practices' sessions, we used mesoshared to connect to a remote server that had the capability of compiling UPC programs.

First, we practised on running our first Hello World program on UPC language using 4 different threads. Then, we coded a conversion table between Fahrenheit and Celsius to understand the functioning of work sharing among threads. After that, we coded 2 vector programs. One that calculates the addition of two vector and another that calculates the multiplication between a vector and a matrix.

We have deepened our knowledges by coding a simplified Laplace solver in UPC, and we finished by coding a 2D heat conduction program.

2 1D Laplace Solver Algorithm

The programs that will follow are the representation of the 1D Laplace Solver Algorithm. Concretely, two tables, named b and x are manipulated to create a new table named x_new. To understand more clearly the operations among tables, the process is represented in the figure below :

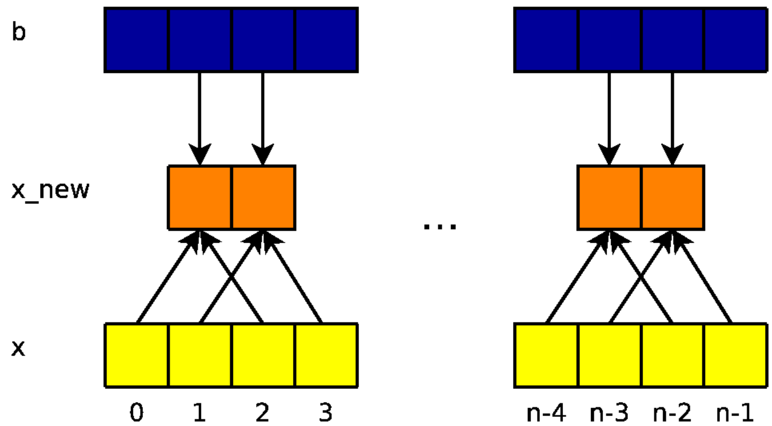


Figure 1: 1D Laplace Solver Algorithm

2.1 1D Laplace Solver in C

First, a C program was given to understand its functioning. The output is displayed at the end of the code.

```
#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;
}
```

```c
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;
    }
}
```

```
[tputbm@mesoshared exercises]$ cc -o ex_2 ex_2.c
[tputbm@mesoshared exercises]$ ./ex_2
   b   |    x   | x_new
==============================
0.9683 | 0.3286 | 0.0000
0.8102 | 0.6814 | 0.8231
0.9664 | 0.5075 | 0.9913
0.2087 | 0.3348 | 0.7676
0.1383 | 0.8191 | 0.5750
0.3980 | 0.6768 | 1.0613
0.3631 | 0.9055 | 0.5671
0.8927 | 0.0943 | 1.1522
0.3393 | 0.5061 | 0.6354
0.4313 | 0.8372 | 0.5586
0.5144 | 0.1798 | 1.0285
0.4182 | 0.7054 | 0.5074
0.3029 | 0.4169 | 0.8739
0.5950 | 0.7395 | 0.8784
0.9514 | 0.7449 | 1.3054
0.0881 | 0.9197 | 0.8657
0.0735 | 0.8983 | 0.5165
0.6012 | 0.0399 | 1.1547
0.4058 | 0.8099 | 0.4949
0.3747 | 0.5441 | 0.9786
0.6290 | 0.7727 | 1.0826
0.2209 | 0.9921 | 0.5536
0.6782 | 0.1136 | 0.8439
0.0863 | 0.0175 | 0.3588
0.6198 | 0.5177 | 0.3857
0.8547 | 0.1342 | 0.8226
0.6974 | 0.2729 | 0.4160
0.8396 | 0.0004 | 0.7736
0.6898 | 0.4346 | 0.6657
0.7399 | 0.6412 | 1.0013
0.1795 | 0.8280 | 0.5369
```

Figure 2: Output of ex_2.c

2.2 1D Laplace Solver in UPC

The C version is working, but it could be improved by using threads. Threads leads to improve the speed of a program and allows using less memory. UPC is a language that allows to use threads.

The ex_3.upc program is an improvement of the ex_2.c one. As a matter of fact, the work among the program is decomposed by each thread. Every thread has a portion of the program attributed.

First, this program's version is using the method of a for loop + condition as follows :

```
for(j = 0; j < (TOTAL\_SIZE)-1; j++)
{
    if(j%THREADS == MYTHREAD)
    {
        ...
    }
}
```

It is working as the tasks are correctly distributed among the threads. But in any case, the for loop is called, and then the if condition is checked, which is not a good way of coding. We will see later a better version of this program.

The upc_barrier is used when it is required that all the threads have finished their execution. Indeed, sometimes, all the threads must have finished their task before the program continues.

The output of the following code is displayed at the end of the code.

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

#define TOTALSIZE 800

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;

    for( j=0; j<(TOTALSIZE)-1; j++ ){
        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;
        }
    }
}
```

```
[tputbm@mesoshared exercises]$ vim ex_3.upc
[tputbm@mesoshared exercises]$ upcc -T=4 ex_3.upc -o ex_3
[tputbm@mesoshared exercises]$ upcrun ex_3
UPCR: UPC thread 0 of 4 on mesoshared (pshm node 0 of 1, process 0 of 4, pid=308624)
UPCR: UPC thread 2 of 4 on mesoshared (pshm node 0 of 1, process 2 of 4, pid=308650)
UPCR: UPC thread 1 of 4 on mesoshared (pshm node 0 of 1, process 1 of 4, pid=308649)
UPCR: UPC thread 3 of 4 on mesoshared (pshm node 0 of 1, process 3 of 4, pid=308651)
   b   |    x    | x_new
=============================
0.3911 | 0.3030 | 0.3631
0.3537 | 0.3351 | 0.3545
0.4668 | 0.0524 | 0.5931
0.2787 | 0.3844 | 0.6460
0.4489 | 0.9609 | 0.8231
0.3825 | 0.8130 | 0.9534
0.9456 | 0.5635 | 0.8958
0.4003 | 0.0330 | 0.5257
0.8504 | 0.0876 | 0.7765
0.5453 | 0.6695 | 0.7994
0.4103 | 0.9659 | 0.9256
0.7934 | 0.7712 | 1.1639
0.5815 | 0.5686 | 1.0592
0.1399 | 0.7657 | 0.5830
0.8688 | 0.4576 | 1.2827
0.6686 | 0.9309 | 0.7535
0.1463 | 0.3808 | 0.6171
0.3462 | 0.1569 | 0.7707
0.8170 | 0.8144 | 0.9366
0.7901 | 0.8993 | 1.0184
0.9140 | 0.4322 | 1.0276
0.8158 | 0.2419 | 0.7375
0.3003 | 0.2270 | 0.6797
0.5245 | 0.8173 | 0.7434
0.2099 | 0.7353 | 0.6507
0.6673 | 0.2743 | 0.9474
0.8252 | 0.4922 | 1.0426
0.3325 | 0.9856 | 0.5855
0.0314 | 0.3462 | 0.8052
0.9423 | 0.5933 | 1.1264
0.6712 | 0.9642 | 1.0140
0.5819 | 0.7635 | 1.2195
0.5772 | 0.8930 | 0.9434
0.7591 | 0.5460 | 0.8649
0.0251 | 0.0776 | 0.2886
0.4433 | 0.0062 | 0.3563
0.4809 | 0.1918 | 0.3559
0.2454 | 0.2247 | 0.4408
0.1534 | 0.4445 | 0.3759
0.6799 | 0.3737 | 0.7617
```

Figure 3: Output of ex_3.upc

2.3 Better work sharing construct with a single for loop

The code below is actually better, as there is only one for loop and no if condition. The for loop is executed by each thread, but all of them are only taking care of their portion of code as the variable is incremented by the number of threads, as displayed below :

```
for(j = MYTHREAD; j < N; j += THREADS)
{
    ...
}
```

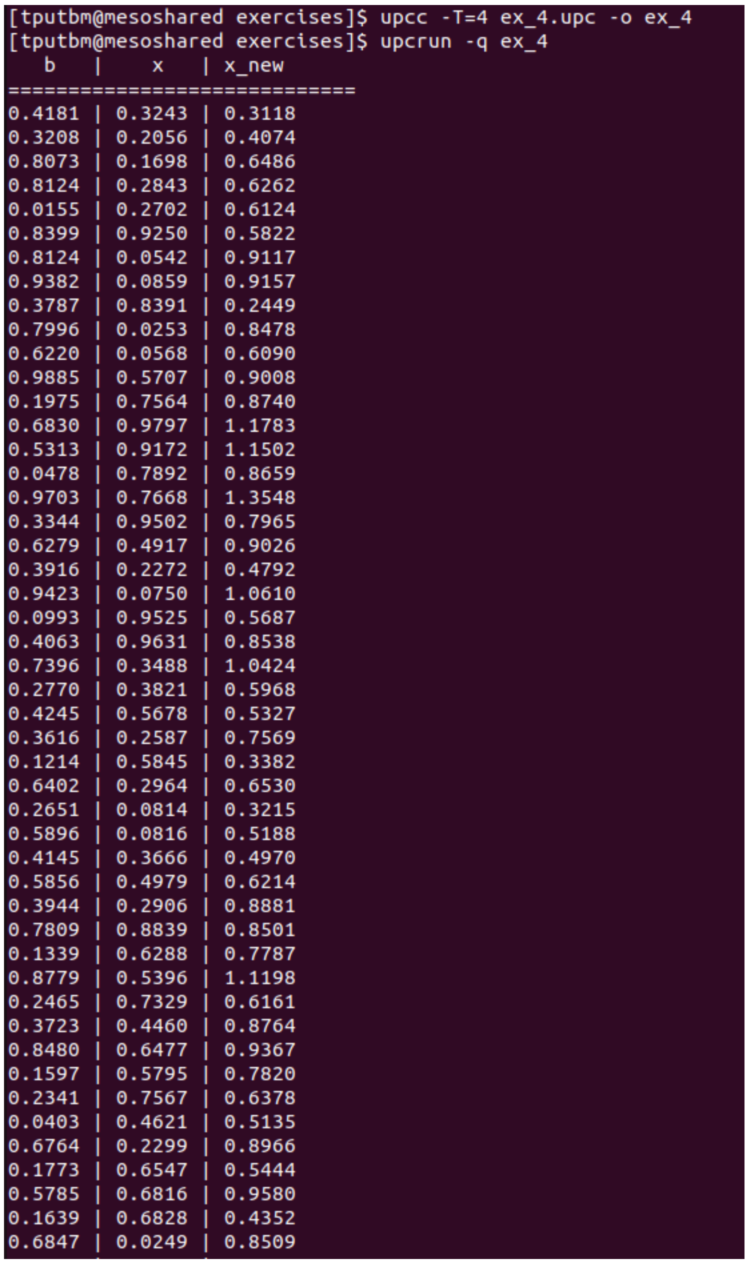
Moreover, UPC has implemented a function that it is equivalent to the loop above, it is named upc_forall and is used as follows :

```
upc_forall(j = 0; j < N; j++; j)
{
    ...
}
```

The output of the code is displayed at the end of the code.

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

#define TOTALSIZE 16
#define N TOTALSIZE*THREADS
shared double x[N];
shared double x_new[N];
shared double b[N];

void init();

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

    init();
    upc_barrier;

    for(j = MYTHREAD; j < N; 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;
        }
    }
}
```



```
[tputbm@mesoshared exercises]$ upcc -T=4 ex_4.upc -o ex_4
[tputbm@mesoshared exercises]$ upcrun -q ex_4
   b   |    x   | x_new
==============================
0.4181 | 0.3243 | 0.3118
0.3208 | 0.2056 | 0.4074
0.8073 | 0.1698 | 0.6486
0.8124 | 0.2843 | 0.6262
0.0155 | 0.2702 | 0.6124
0.8399 | 0.9250 | 0.5822
0.8124 | 0.0542 | 0.9117
0.9382 | 0.0859 | 0.9157
0.3787 | 0.8391 | 0.2449
0.7996 | 0.0253 | 0.8478
0.6220 | 0.0568 | 0.6090
0.9885 | 0.5707 | 0.9008
0.1975 | 0.7564 | 0.8740
0.6830 | 0.9797 | 1.1783
0.5313 | 0.9172 | 1.1502
0.0478 | 0.7892 | 0.8659
0.9703 | 0.7668 | 1.3548
0.3344 | 0.9502 | 0.7965
0.6279 | 0.4917 | 0.9026
0.3916 | 0.2272 | 0.4792
0.9423 | 0.0750 | 1.0610
0.0993 | 0.9525 | 0.5687
0.4063 | 0.9631 | 0.8538
0.7396 | 0.3488 | 1.0424
0.2770 | 0.3821 | 0.5968
0.4245 | 0.5678 | 0.5327
0.3616 | 0.2587 | 0.7569
0.1214 | 0.5845 | 0.3382
0.6402 | 0.2964 | 0.6530
0.2651 | 0.0814 | 0.3215
0.5896 | 0.0816 | 0.5188
0.4145 | 0.3666 | 0.4970
0.5856 | 0.4979 | 0.6214
0.3944 | 0.2906 | 0.8881
0.7809 | 0.8839 | 0.8501
0.1339 | 0.6288 | 0.7787
0.8779 | 0.5396 | 1.1198
0.2465 | 0.7329 | 0.6161
0.3723 | 0.4460 | 0.8764
0.8480 | 0.6477 | 0.9367
0.1597 | 0.5795 | 0.7820
0.2341 | 0.7567 | 0.6378
0.0403 | 0.4621 | 0.5135
0.6764 | 0.2299 | 0.8966
0.1773 | 0.6547 | 0.5444
0.5785 | 0.6816 | 0.9580
0.1639 | 0.6828 | 0.4352
0.6847 | 0.0249 | 0.8509
```

Figure 4: Output of ex_4.upc

2.4 Blocked arrays and work sharing with upc_forall

The ex_5 version of this code actually uses the upc_forall() loop, we can see it below. The output of the code is displayed at the end of the code.

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

#define BLOCKSIZE 16
#define N BLOCKSIZE*THREADS

shared [BLOCKSIZE] double x[N];
shared [BLOCKSIZE] double x_new[N];
shared [BLOCKSIZE] double b[N];

void init();

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

    init();
    upc_barrier;

    upc_forall( j=0; j<(N)-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<N; 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<N; i++ ){
            b[i] = (double)rand() / RAND_MAX;
            x[i] = (double)rand() / RAND_MAX;
        }
    }
}
```

```
[tputbm@mesoshared exercises]$ upcc -T=4 ex_5.upc -o ex_5
[tputbm@mesoshared exercises]$ upcrun -q ex_5
   b   |    x    | x_new
=============================
0.4469 | 0.9649 | 0.2505
0.8971 | 0.0542 | 1.3027
0.7418 | 0.7434 | 0.5715
0.7005 | 0.3471 | 1.2197
0.1779 | 0.9954 | 0.2814
0.8394 | 0.0379 | 1.2245
0.1089 | 0.6141 | 0.4303
0.3755 | 0.7138 | 0.9115
0.5467 | 0.8333 | 0.7096
0.1372 | 0.1588 | 0.5488
0.3893 | 0.1272 | 0.5455
0.8112 | 0.5429 | 0.8030
0.8643 | 0.6677 | 1.1697
0.5612 | 0.9322 | 1.0024
0.8094 | 0.7760 | 0.9905
0.3061 | 0.2395 | 0.7233
0.6862 | 0.3645 | 0.5769
0.6906 | 0.2280 | 1.0216
0.0906 | 0.9881 | 0.4112
0.7240 | 0.5038 | 1.0952
0.8860 | 0.4782 | 1.1547
0.0229 | 0.9197 | 0.3871
0.3367 | 0.2731 | 1.0345
0.1550 | 0.8126 | 0.3065
0.8103 | 0.1849 | 1.1140
0.1190 | 0.6051 | 0.1684
0.7973 | 0.0330 | 1.0337
0.0058 | 0.6650 | 0.0546
0.3857 | 0.0705 | 0.9293
0.1509 | 0.8079 | 0.3768
0.6122 | 0.5323 | 0.8927
0.0002 | 0.3653 | 0.6251
0.6808 | 0.7177 | 0.7697
0.6733 | 0.4934 | 0.9079
0.2347 | 0.4249 | 0.3925
0.1063 | 0.0569 | 0.3023
0.2281 | 0.0734 | 0.3596
0.3075 | 0.4343 | 0.5222
0.4761 | 0.6635 | 0.6743
0.8511 | 0.4382 | 1.0835
0.3521 | 0.6523 | 0.8704
0.2955 | 0.9506 | 0.7648
0.0182 | 0.5818 | 0.5310
0.8286 | 0.0933 | 1.0711
0.8637 | 0.7318 | 0.8838
0.5582 | 0.8106 | 0.8608
0.8007 | 0.4315 | 1.1320
```

Figure 5: Output of ex_5.upc

2.5 Synchronization

Finally, this program's version allows realizing the procedure many times. Here, it is done 10000 times.

This is possible thanks to the copying of the x_new table into x table after that the x_new table is created. It is done directly by value, i.e. that every value of the x_new table is copied into x table. One way to improve the efficiency of the program would be to use pointers, which consumes less memory.

```
#include <upc_relaxed.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define BLOCKSIZE 16

#define N BLOCKSIZE*THREADS

shared [BLOCKSIZE] double x[N];
shared [BLOCKSIZE] double x_new[N];
shared [BLOCKSIZE] double b[N];

void init();

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

    init();
    upc_barrier;

    for( iter=0; iter<10000; iter++ ){
        upc_forall( j=1; j<N-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<N; 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<N; 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<N; i++ ){
            b[i] = (double)rand() / RAND_MAX;
            x[i] = (double)rand() / RAND_MAX;
        }
    }
}
```

```
[tputbm@mesoshared exercises]$ upcc -T=4 ex_6.upc -o ex_6
[tputbm@mesoshared exercises]$ upcrun -q ex_6
   b    |     x    | x_new
==============================
0.4616 | 0.0000 | 0.0000
0.0918 | 14.6216 | 14.6216
0.2130 | 29.1514 | 29.1514
0.4065 | 43.4682 | 43.4682
0.4634 | 57.3785 | 57.3785
0.4840 | 70.8254 | 70.8254
0.2323 | 83.7883 | 83.7883
0.7516 | 96.5189 | 96.5189
0.3927 | 108.4979 | 108.4979
0.8206 | 120.0842 | 120.0842
0.3451 | 130.8500 | 130.8500
0.9573 | 141.2706 | 141.2706
0.6816 | 150.7338 | 150.7338
0.3089 | 159.5155 | 159.5155
0.4561 | 167.9882 | 167.9882
0.1140 | 176.0049 | 176.0049
0.6003 | 183.9075 | 183.9075
0.5082 | 191.2099 | 191.2099
0.4360 | 198.0041 | 198.0041
0.1181 | 204.3622 | 204.3622
0.3930 | 210.6023 | 210.6023
0.8717 | 216.4493 | 216.4493
0.8962 | 221.4246 | 221.4246
0.5978 | 225.5036 | 225.5036
0.8336 | 228.9849 | 228.9849
0.4053 | 231.6326 | 231.6326
0.1725 | 233.8750 | 233.8750
0.1898 | 235.9449 | 235.9449
0.7074 | 237.8249 | 237.8249
0.1673 | 238.9977 | 238.9977
0.8667 | 240.0031 | 240.0031
0.1748 | 240.1418 | 240.1418
0.1937 | 240.1057 | 240.1057
0.7503 | 239.8760 | 239.8760
0.9588 | 238.8960 | 238.8960
0.2168 | 236.9571 | 236.9571
0.2451 | 234.8014 | 234.8014
0.1985 | 232.4007 | 232.4007
0.3329 | 229.8014 | 229.8014
0.0989 | 226.8693 | 226.8693
0.8366 | 223.8383 | 223.8383
0.0402 | 219.9707 | 219.9707
```

Figure 6: Output of ex_6.upc

2.6 Reduction operation

The code that follows allows determining the maximum difference between x and x_new tables. In order to determine if there is convergence, or not, UPC owns a function named upc_all_reduceD() that is used in the program below. The output is as always, written at the end of this code.

```
#include <upc.h>
#include <upc_collective.h>
#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;
    diffmax = 0.0;
    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_OUT_ALLSYNC);
        printf("diff␣max␣=␣%lf␣\n", diffmax);

        upc_barrier;

        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;
    }

    if( MYTHREAD == 0 ){
        for(j=0; j<TOTALSIZE*THREADS; j++){
            printf("%f\t", x_new[j]);
        }
        printf("\n");
    }

    return 0;
}
```

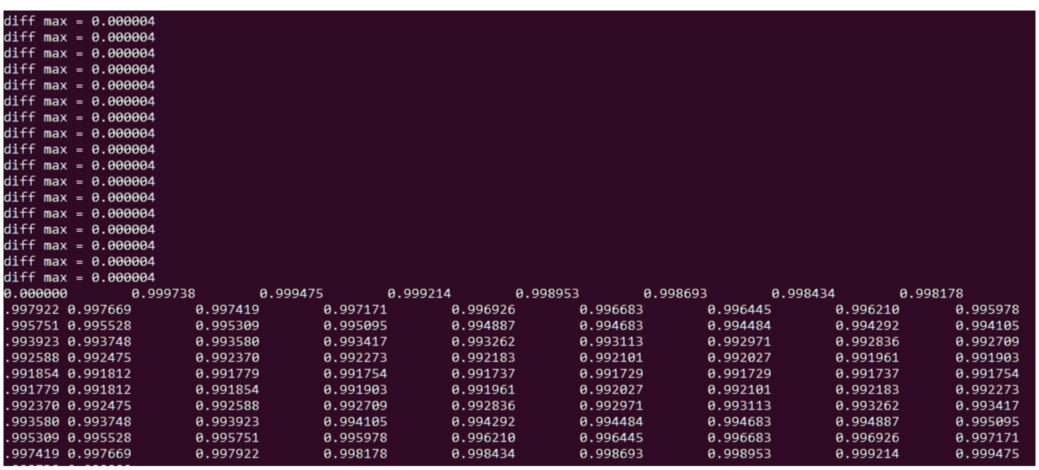


Figure 7: Output of ex_7.upc

3 2D Heat

3.1 Heat C

At first, we began by launching a first 2D heat program in C language. As C language does not support threads, the time of the program to iterate every index in the grid table is obviously longer than any program coded in UPC using threads. As a matter of fact, this program only travels the grid table and copies the grid table into a new_grid.

```
#include <stdio.h>
#include <math.h>
#include <sys/time.h>

#define N 598

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

void initialize()
{
    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()
{
    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++ )
                for(l = 0; l < N + 2; l++ )
                    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␣%.4lf␣sec\n", nr_iter, time);
    printf("Time␣per␣iteration␣:␣%.4lf␣\n", time * 1000. / nr_iter);

    return 0;
}
```

3.2 Heat 1

Now that C version of the 2D heat mechanism has been introduced, we were able to code it using UPC and threads. The main goal of this new version of the program was to distribute the different tasks between threads, essentially by calculating the T variable and copying the new_grid table thanks to upc_forall command. As all threads owns his portion of code dedicated in each loop, the program runs way faster than the previous version.

```
#include <stdio.h>
#include <math.h>
#include <sys/time.h>

#define N 902
#define BLOCK_SIZE ((N+2)*(N+2)/THREADS)

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

void initialize()
{
    int j;

    /* Heat one side of the solid */
```

```
    if(MYTHREAD == 0){
        for(j = 1; j< N + 1; j++)
        {
            grid[0][j] = 1.0;
            new_grid[0][j] = 1.0;
        }
    }
}

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

    initialize();

    upc_barrier;

    /* 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;
        upc_forall(i = 1; i< N + 1; i++; 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 */
            }
        }
        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
        {
            upc_forall(k = 0; k < N + 2; k++; k){
                for(l = 0; l < N + 2; l++){
                    grid[k][l] = new_grid[k][l]; /* iteration */
                }
            }
        }
        upc_barrier;
        nr_iter++;
    } while( finished == 0 );

    upc_barrier;

    if(MYTHREAD == 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);
        printf("Time per iteration : %.4lf \n", time * 1000. / nr_iter);
    }
    return 0;
}
```

3.3 Heat 3

As passage by value is a really heavy process, indeed, it takes a lot of resources for the program to copy a table to another. One way to avoid this in C and UPC is using pointers. Since every pointer points to an address to memory, manipulation of them requires way less resources than directly manipulating data.

Here, we are using two pointers, *ptr[N+2] that points to grid table and *new_ptr[N+2] that points to new_grid table. Thanks to that, the only main difference between this new program below and the latest is that we only manipulate these pointers that are pointing each on an index of its dedicated table fixed at the beginning of the program.

Thus, this new version is way faster than the previous one.

```
#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include <upc_relaxed.h>
#include <upc_collective.h>

#define N 598
#define BLOCK_SIZE ((N+2) * (N+2) / THREADS)

shared[BLOCK_SIZE] double grid[N+2][N+2];
shared[BLOCK_SIZE] double new_grid[N+2][N+2];
shared[BLOCK_SIZE] double *ptr[N+2];
shared[BLOCK_SIZE] double *new_ptr[N+2];
shared[BLOCK_SIZE] double* temp;
```

```
shared double dTmax_local[THREADS];

void initialize()
{
    int i, j;

    if(MYTHREAD == 0)
    {
        for(i = 0; i < N + 2; i++)
        {
            grid[0][i] = 1.0;
            new_grid[0][i] = 1.0;
        }
    }
}

int main() {

    struct timeval ts_st, ts_end;

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

    initialize();

    upc_barrier;

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

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

    upc_barrier;

    gettimeofday(&ts_st, NULL);

    do
    {
        dTmax = 0.0;

        upc_forall(i = 1; i < N + 1; i++; i)
        {
            for(j = 1; j < N + 1; j++)
            {
                T = 0.25 * (ptr[i + 1][j]
                + ptr[i-1][j] + ptr[i][j - 1]
                + ptr[i][j + 1]);
```

```
                dT = T - ptr[i][j];
                new_ptr[i][j] = T;
                 if(dTmax < fabs(dT) )
                    dTmax = fabs(dT);
            }
        }

        dTmax_local[MYTHREAD] = dTmax;

        upc_barrier;

        dTmax = dTmax_local[0];

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

        upc_barrier;

        if( dTmax < epsilon)
            finished = 1;
        else
        {

            for(k = 0; k < N + 2; k++)
            {
                temp = ptr[k];
                ptr[k] = new_ptr[k];
                new_ptr[k] = temp;
           }
        }

        nr_iter++;

    } while(finished == 0);

    upc_barrier;

    gettimeofday(&ts_end, NULL);

    if(MYTHREAD == 0)
    {
        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␣%.4lf␣sec\n", nr_iter, time);
        printf("time␣per␣iteration␣:␣%.4lf␣\n",time * 1000. / nr_iter);
    }

    return 0;
}
```

3.4 Heat 4

Another way to boost performance is the use of private pointers. Indeed, execution among private pointers is way faster than manipulating them in the shared space. This notion allows us to improve our latest program by using new private pointers called *ptr_priv[N_PRIV] and *new_ptr_priv[N_PRIV] that will carry out a large part of the process of the program.

The data displayed at the end of the code actually shows how this new version is way faster than the heat_3 one.

```
#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include <upc_relaxed.h>

#define N 598
#define SIZE ((N+2) * (N+2) / THREADS)
#define N_PRIV ((N+2)/THREADS)

shared[SIZE] double grid[N+2][N+2];
shared[SIZE] double new_grid[N+2][N+2];
shared[SIZE] double *ptr[N+2];
shared[SIZE] double *new_ptr[N+2];
shared[SIZE] double *temp;

double *ptr_priv[N_PRIV];
double *new_ptr_priv[N_PRIV];
double *temp_priv;

shared double dTmax_local[THREADS];

void initialize()
{
    int j, k, l;

     if(MYTHREAD == 0)
    {
        for(j = 1; j < N + 2; j++)
        {
            grid[0][j] = 1.0;
            new_grid[0][j] = 1.0;
        }
    }
}

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

    initialize();
```

```
    upc_barrier;

    /* Set the precision wanted */
    epsilon  = 0.0001;
    finished = 0;
    nr_iter = 0;

     /* Initilisation of ptr and new_ptr */
    for(i = 0; i < N + 2; i++)
    {
        ptr[i] = &grid[i][0];
        new_ptr[i] = &new_grid[i][0];
    }

    /* Initialisation of private pointers */

    for(i = 0; i < N_PRIV; i++)
    {
        ptr_priv[i] = (double *) &grid[i + (MYTHREAD * N_PRIV)][0];
        new_ptr_priv[i] = (double *) &new_grid[i + (MYTHREAD * N_PRIV)][0];
    }

    upc_barrier;

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

    do
    {
        dTmax = 0.0;
        i = 0;
        //dTmax_local[MYTHREAD] = 0.0;

         if(i + N_PRIV*MYTHREAD > 0) {
            for( j=1; j < N; j++) {
                T = 0.25 * (ptr_priv[1][j]
                + ptr[MYTHREAD*N_PRIV + i - 1][j]
                + ptr_priv[0][j - 1] + ptr_priv[0][j + 1]);
                dT = T - ptr_priv[0][j];

                new_ptr_priv[0][j] = T;

                if( dTmax < fabs(dT) )
                    dTmax = fabs(dT);
            }
        }

        for(i += 1; i < N_PRIV - 1; i++) {
            for(j = 1; j < N; j++) {
                T = 0.25 * (ptr_priv[i + 1][j] + ptr_priv[i - 1][j]
                + ptr_priv[i][j - 1] + ptr_priv[i][j + 1]);
                dT = T - ptr_priv[i][j];

                new_ptr_priv[i][j] = T;
```

```
            if( dTmax < fabs(dT) )
                dTmax = fabs(dT);
        }
    }

    // Last row
    if(i + N_PRIV*MYTHREAD < N + 1) {
        for(j = 1; j < N; j++)
        {
            T = 0.25 * (ptr[N_PRIV * MYTHREAD + i + 1][j]
            + ptr_priv[i - 1][j]
            + ptr_priv[i][j - 1] + ptr_priv[i][j + 1]);
            dT = T - ptr_priv[i][j];

            new_ptr_priv[i][j] = T;

            if( dTmax < fabs(dT) )
                dTmax = fabs(dT);
        }

    }

    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];
    }

    if( dTmax < epsilon ) /* is the precision reached good enough ? */
        finished = 1;
    else
    {
        for(k = 0; k < N + 2; k++){ /* not yet ... Need to prepare */
            temp = ptr[k];
            ptr[k] = new_ptr[k];
            new_ptr[k] = temp;
        }

        for(k = 0; k < N_PRIV; k++){
            temp_priv = ptr_priv[k];
            ptr_priv[k] = new_ptr_priv[k];
            new_ptr_priv[k] = temp_priv;
        }
    }
    nr_iter++;

} while(finished == 0);

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

if(MYTHREAD == 0)
{
```

```
        /* 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 %.4lf sec\n", nr_iter, time);
        printf("Time per iteration : %.4lf \n", time * 1000. / nr_iter);
    }

    return 0;
}
```

3.5 Heat 5

During all the measurements of the old programs, we needed to specify the N size on the code itself. In order to specify it on the terminal, the use of dynamic allocation was necessary in order to allocate size for the pointers of our program, as the N size is no longer defined in the code.

The upc_all_alloc() function is used to allocated dynamically memory.

```
#include <upc_relaxed.h>
#include <upc_collective.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <sys/time.h>

shared double dTmax_local[THREADS];

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

    for(j = 0; j < N + 2; j++)
    {
        grid[j] = 1.0;
        new_grid[j] = 1.0;
    }
}

int main(int argc, char* argv[])
{
    struct timeval ts_st, ts_end;
    double dTmax, dT, epsilon, time;
    int finished, i, j, k, l;
    double T;
    int nr_iter;
```

```
if(argc != 2)
{
      printf("Enter␣2␣arguments␣please␣\n");
    return -1;
}

int N = atoi(argv[1]);
int BLOCK_SIZE = ((N + 2) * (N + 2) / THREADS);

shared[] double* ptr = upc_all_alloc(BLOCK_SIZE,
                                        BLOCK_SIZE * THREADS);
shared[] double* new_ptr = upc_all_alloc(BLOCK_SIZE,
                                        BLOCK_SIZE * THREADS);

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

if(MYTHREAD == 0)
{
    initialize(ptr, new_ptr, N);
}

upc_barrier;

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

gettimeofday(&ts_st, NULL);

do
{
    dTmax = 0.0;
    int iMax = (N + 2) * (MYTHREAD + 1) / THREADS;

    if(iMax > N + 1)
        iMax = N + 1;

    upc_forall(i = 1; i < iMax; i++; i)
    {
         for(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 = T - ptr_get(i, j);

            new_ptr[i * (N + 2) + j] = T;

            if(dTmax < fabs(dT))
                dTmax = fabs(dT);
        }
    }
```

```
        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)
        {
            finished = 1;
        }
        else
        {
            shared[] double* temp;
            temp = ptr;
            ptr = new_ptr;
            new_ptr = temp;
        }

        nr_iter++;

    }while(finished == 0);

    gettimeofday(&ts_end, NULL);

    if(MYTHREAD == 0)
    {
        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 %.4lf sec\n", nr_iter, time);
        printf("Time per iteration : %.4lf \n", time * 1000. / nr_iter);
    }

    return 0;
}
```

3.6 Analysis of the results

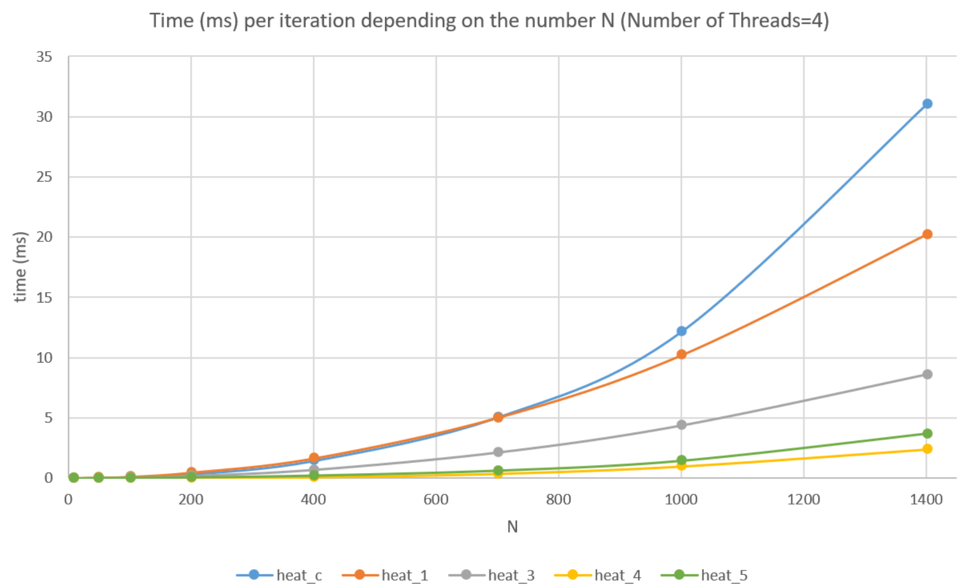


Figure 8: Evolution of time per iteration depending on the number N

T = 4		heat_c	heat_1	heat_3	heat_4	heat_5
		time (ms)				
N	10	0,001	0,0062	0,0032	0	0,0087
	50	0,0249	0,0522	0,0318	0,0092	0,0251
	102	0,1024	0,1229	0,0532	0,0154	0,0483
	202	0,3306	0,4736	0,1947	0,0208	0,0902
	402	1,4504	1,6576	0,7079	0,0826	0,2388
	702	5,0598	5,0309	2,1423	0,3719	0,6434
	1002	12,1913	10,2273	4,3821	0,9835	1,4559
	1402	31,0458	20,2405	8,5961	2,3926	3,6823

Figure 9: [TABLE] Evolution of time per iteration depending on the number N

In order to measure the efficiency of every program, the time needed for 1 iteration has been written. The higher the time per iteration is high, the slower the program is.

These results are not so surprising. As a matter of fact, heat_c is the slowest version as it is only running on 1 thread. On the other side, heat_4, which is a UPC_version, can run on many threads, and then improving performances of the program.

Moreover, all our expectations are verified : heat_1 is an improved version of heat_c, which is a bit better than its C version. Finally, heat_3, which was the first program of 2D Heat that used the concept of pointers, is as it shows on the graph way better than directly manipulating tables like the heat_1 version.

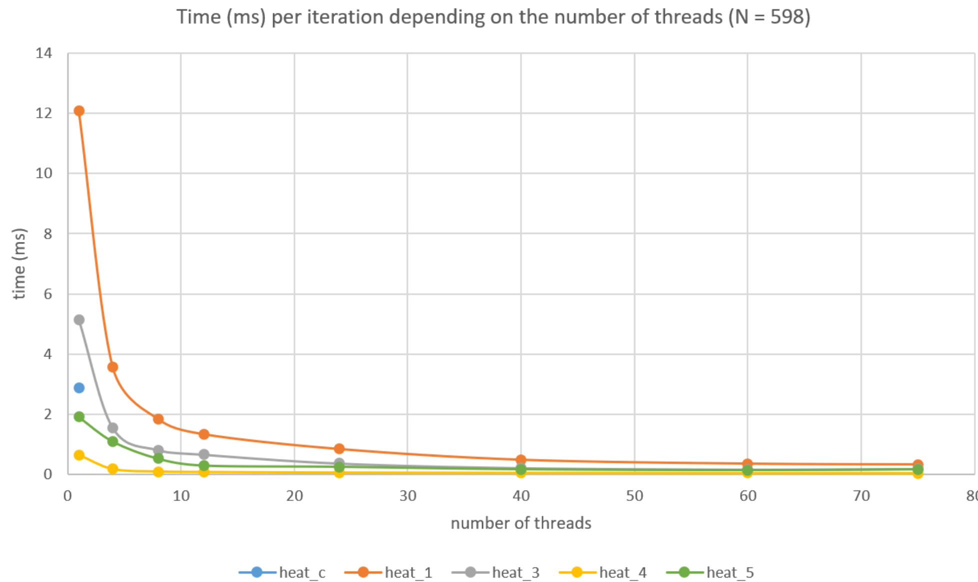


Figure 10: Evolution of time per iteration depending on the number of threads

N = 598		heat_c	heat_1	heat_3	heat_4	heat_5
		time (ms)				
T	1	2,8796	12,0877	5,1469	0,6494	1,9157
	4		3,554	1,545	0,1743	1,0947
	8		1,8356	0,8071	0,0915	0,5202
	12		1,3316	0,6555	0,0762	0,2924
	24		0,8389	0,3619	0,0577	0,2517
	40		0,4834	0,2097	0,0463	0,1703
	60		0,3545	0,1591	0,0399	0,1452
	75		0,3307	0,1752	0,0358	0,1714

Figure 11: [TABLE] Evolution of time per iteration depending on the number of threads

This graph above also shows that when the number of threads increases, the program is executed way faster. Moreover, our programs are all at the same position as the previous graphic. Thus, the most efficient program is the heat_4 one, followed very closely by heat_5. heat_3 is still a very efficient program when the number of threads increases, but heat_1 is the worst with more than 12 ms per iteration when the number of threads is equal to 1.

Below is a graph to see more clearly the difference of speed among the programs, by removing the first line "threads = 1", as the heat_1 version is particularly slow and then squashed the graph.

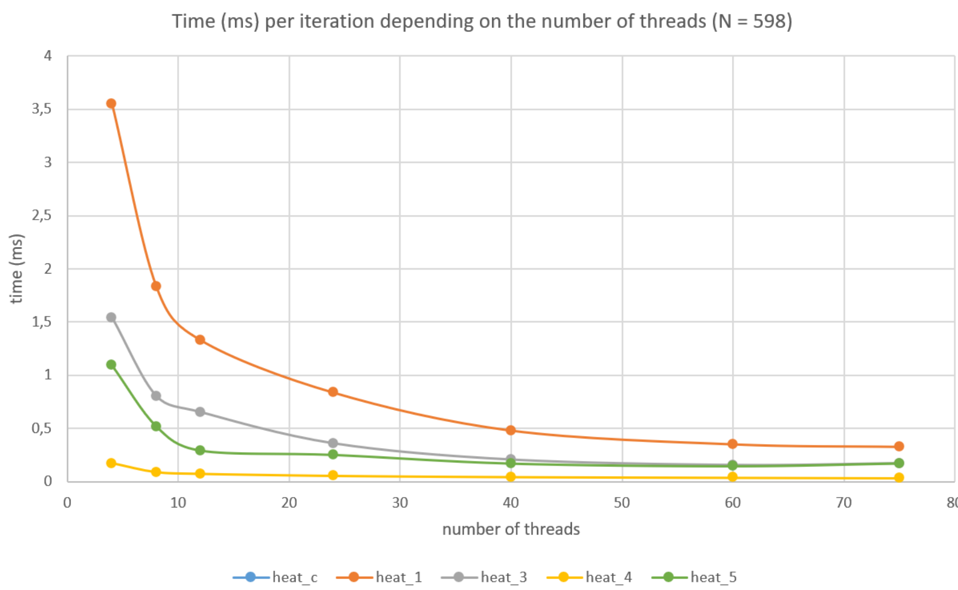


Figure 12: Evolution of time per iteration depending on the number of threads, zoomed