UPC

GIT repository : https://github.com/AurelienVasseur/LO47_UPC_HandsOn

---

3 - Simplified 1D Laplace solver

3.1 - A simplified 1-D Laplace solver in C

This first 1-D Laplace solver in C is not optimized. Indeed, We don't use the threads power to increase the program performance. The program do all iterations. In fact, there is no work sharing between different threads.

3.2 - The 1D solver in UPC

In this solution we are going to share work between many threads. To do this, we start by create shared variables in the shared space. These variables will be used by all of the threads.

```
8   //== declare the x, x_new, b arrays in the shared space with size of TOTALSIZE
9   shared double x[TOTALSIZE];
10  shared double x_new[TOTALSIZE];
11  shared double b[TOTALSIZE];
12
13  void init();
14
15  int main(int argc, char **argv){
16      int j;
17
18      init();
19      upc_barrier;
20
21      //== add a for loop which goes through the elements in the x_new array
22      for( j=1; j<(TOTALSIZE)-1; j++ ){
23          //== insert an if statement to do the work sharing across the threads
24          if( (j%THREADS) == MYTHREAD){
25              x_new[j] = 0.5*( x[j-1] + x[j+1] + b[j] );
26          }
27      }
```

To share the work we divide it inside the loop which goes through the elements in the x_new array. To do this, we add a if statement which will allow each thread to test if it is up to him to do this task. This is a first way to do the work sharing.

```
21      //== add a for loop which goes through the elements in the x_new array
22      for( j=1; j<(TOTALSIZE)-1; j++ ){
23          //== insert an if statement to do the work sharing across the threads
24          if( (j%THREADS) == MYTHREAD){
25              x_new[j] = 0.5*( x[j-1] + x[j+1] + b[j] );
26          }
27      }
28
29      if( MYTHREAD == 0 ){
30          printf("   b   |    x   | x_new\n");
31          printf("============================\n");
32
33          for( j=0; j<TOTALSIZE; j++ )
34              printf("%1.4f | %1.4f | %1.4f \n", b[j], x[j], x_new[j]);
35      }
```

After that, we start printing the result on the screen. To do this work we will only use the thread 0. This is why there is a if statement which check if it is thread 0.

With this solution we will meet a problem. Thread 0 may start printing result before others finish. The problem is that there is no condition which force threads to expect them between the end of the main work (l. 27) and the start of the printing (l. 29). If we want to start printing result only after all threads finished their work we need to add a upc_barrier.

```
21      //== add a for loop which goes through the elements in the x_new array
22      for( j=1; j<(TOTALSIZE)-1; j++ ){
23          //== insert an if statement to do the work sharing across the threads
24          if( (j%THREADS) == MYTHREAD){
25              x_new[j] = 0.5*( x[j-1] + x[j+1] + b[j] );
26          }
27      }
28
29      upc_barrier;
30
31      if( MYTHREAD == 0 ){
32          printf("   b   |    x   | x_new\n");
33          printf("============================\n");
34
35          for( j=0; j<TOTALSIZE; j++ )
36              printf("%1.4f | %1.4f | %1.4f \n", b[j], x[j], x_new[j]);
37      }
```

3.3 - Better work sharing construct with a single for loop

In this solution we will improve the for loop to perform the work distribution across threads.

```
20      // ==> setup j to point to the first element so that the current thread should
21      // progress (in respect to its affinity)
22      j = MYTHREAD;
23
24      // ==> add a for loop which goes only through the elements in the x_new array
25      // with affinity to the current THREAD
26
27      for( j=MYTHREAD+1; j<(TOTALSIZE*THREADS)-1; j+=THREADS )
28          x_new[j] = 0.5*( x[j-1] + x[j+1] + b[j] );
29
30      upc_barrier;
31
32      if( MYTHREAD == 0 ){
33          printf("   b   |    x   | x_new\n");
34          printf("============================\n");
35
36          for( j=0; j<TOTALSIZE*THREADS; j++ )
37              printf("%1.4f | %1.4f | %1.4f \n", b[j], x[j], x_new[j]);
38      }
```

To do that we will edit the for loop which was coupled with a if statement in the previous solution. In fact we will delete the if statement and update the loop. In the previous solution each thread do each iteration of the loop (which correspond of the array size) even if it was to him to do the work. Now, each thread do only iterations which match with his work.

3.4 - Blocked arrays and work sharing with upc_forall

In this new solution we will use the upc_forall power which help us to share work between many threads. In fact, upc_forall allows the distribution of the iterations according to the affinity.

```
6   #define BLOCKSIZE 16
7
8   //==> declare the x, x_new and b arrays in the shared space with size of
9   //    BLOCKSIZE*THREADS and with blocking size of BLOCKSIZE
10  shared [BLOCKSIZE] double x[BLOCKSIZE*THREADS];
11  shared [BLOCKSIZE] double x_new[BLOCKSIZE*THREADS];
12  shared [BLOCKSIZE] double b[BLOCKSIZE*THREADS];
13
14  void init();
15
16  int main(int argc, char **argv){
17      int j;
18
19      init();
20      upc_barrier;
21
22      //==> insert a upc_forall statement to do work sharing while
23      //    respecting the affinity of the x_new array
24      upc_forall( j=1; j<(BLOCKSIZE*THREADS)-1; j++; j){
25          x_new[j] = 0.5*( x[j-1] + x[j+1] + b[j] );
26      }
27      upc_barrier;
```

3.5 - Synchronization

```
20      // add two barrier statements, to ensure all threads finished computing
21      // x_new[] and to ensure that all threads have completed the array
22      // swapping.
23      for( iter=0; iter<10000; iter++ ){
24          upc_forall( j=1; j<BLOCKSIZE*THREADS-1; j++; &x_new[j] ){
25              x_new[j] = 0.5*( x[j-1] + x[j+1] + b[j] );
26          }
27          upc_barrier;
28          upc_forall( j=0; j<BLOCKSIZE*THREADS; j++; &x_new[j] ){
29              x[j] = x_new[j];
30          }
31          upc_barrier;
32      }
```

In this new example we want to synchronize all of the threads. To do that we added two barrier statements which force threads to expect them. These barrier are two upc_barrier (l.27 & l.31).

3.6 - Reduction operation

In this last example we want to keep track of the maximum difference between the x and x_new. To begin, each thread need to compute a local version of the difference. After that, we need to compare all of these values to find which is the biggest.

```
41      upc_forall( j=1; j<TOTALSIZE*THREADS-1; j++; &x_new[j] ){
42          x_new[j] = 0.5 * ( x[j-1] + x[j+1] + b[j] );
43
44          if( diff[MYTHREAD] < x_new[j] - x[j] )
45              diff[MYTHREAD] = x_new[j] - x[j];
46      }
47
48      // Each thread as a local value for diff
49      // The maximum of those values should be used to check
50      // the convergence.
51      diffmax = diff[0];
52      for(i=1; i<THREADS; i++)
53      {
54          if(diffmax < diff[i])
55              diffmax = diff[i];
56      }
```

Indeed, we need to browse each thread diff value. To do that we use a for loop (l.52). Inside the loop we check with a if statement if the diff value of the thread is the biggest. If is true we save this value on diffmax.

4 - 2D Heat conduction

4.4 - First UPC program

In this part I created the UPC version of the 2D Heat conduction program. The following program is not optimized, this is only a conversion from C language to UPC.

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

#define N 30
#define BLOCKSIZE N

shared[BLOCKSIZE] double grids[2][N][N][N];
shared double dTmax_local[THREADS];

void initialize(void)
{
    int y, x;
    for(y=1; y<N-1; y++) {
        upc_forall(x=1; x<N-1; x++; &grid[0][0][y][x]) {
            grids[0][0][y][x] = grids[1][0][y][x] = 1.0;
        }
    }
}

int main(void)
{
    double dTmax, dT, epsilon;
    int finished, z, y, x, i;
    double T;
    int nr_iter;
    int sg, dg;

    initialize();
```

```
32      /* set the constants */
33      epsilon = 0.0001;
34      finished = 0;
35      nr_iter = 0;
36      sg = 0;
37      dg = 1;
38
39      /* synchronization */
40      upc_barrier;
41
42      do
43      {
44          dTmax = 0.0;
45          for(z=1; z<N-1; z++) {
46              for(y=1; y<N-1; y++) {
47                  upc_forall(x=1; x<N-1; x++: &grids[sg][z][y][x]) {
48                      T = (grids[sg][z+1][y][x] + grids[sg][z-1][y][x]
49                      + grids[sg][z][y-1][x] + grids[sg][z][y+1][x]
50                      + grids[sg][z][y][x-1 + grids[sg][z][y][x+1]]) / 6.0;
51                      dT = T - grids[sg][z][y][x];
52                      grids[dg][z][y][x] = T;
53                      if (dTmax < fabs(dT))
54                      {
55                          dTmax + fabs(dT);
56                      }
57                  }
58              }
59          }
60
61          dTmax_local[MYTHREAD] = dTmax;
62          upc_barrier;
```

```
63
64          dTmax = dTmax_local[0];
65          for(i=1; i<THREADS; i++) {
66              if (dTmax < dTmax_local[i]){
67                  dTmax = dTmax_local[i];
68              }
69
70          }
71          upc_barrier;
72
73          if(dTmax < epsilon) {
74              finished = 1;
75          } else {
76              dg = sg;
77              sg = !sg;
78          }
79          nt_iter++;
80      } while (!finsihed);
81
82      upc_barrier;
83
84      if(MYTHREAD == 0) {
85          printf("%d iterations \n", nr_iter);
86      }
87
88      return 0;
89  }
90
```

4.5 - Better memory use

In this second version of the program, I optimized the memory use. Indeed, the previous version is copying the complete grid_new[][] to grid[] at the end of each iteration. This step creates an overhead and can be avoided by implementing a pointer flipping. This is what I implemented in this new program.

Now, the grids grid[][] and grid_new[][] will be accessed using either ptr[][] or new_ptr[][], each respectively pointing to grid[][] and grid_new[][]. At the end of an iteration, instead of copying the complete grid_new[][] to grid[] (like the previous version), the pointers ptr[][] and new_ptr[][] are swapped (l. 111-116) and a new iteration can take place.

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

#define N 30

/* ------------------------------------------------------------ */
/* Complete the grids declarations to be shared and blocked as the
   biggest chunk of rows */
shared [N+2] double grid[N+2][N+2], new_grid[N+2][N+2];
/* add a new shared array, in which each thread will maintain its local
   execution time */
shared double exectime[THREADS];
/* now declare dTmax_local, array of THREADS elements, as shared. This
   array is used to reduce dTmax across THREADS */
shared double dTmax_local[THREADS];
/* ------------------------------------------------------------ */


/* ------------------------------------------------------------ */
/* Declare two new arrays of shared pointers, having same block sizes as
 * grid[][] and new_grid[][] - The dimension of each pointer can be only N+2
 * since each pointer only needs to point to the beginning of each row of
 * grid[][] or new_grid[][].
   A single shared pointer can also be declared for temporary use in the
   pointer swapping operation (we mean by shared arrays of pointers here
   private arrays of pointers, each element pointing to a shared area)*/
shared [N+2] double *ptr[N+2], *new_ptr[N+2], *tmp_ptr;
/* ------------------------------------------------------------ */

```

```
32  void initialize(void)
33  {
34      int j;
35
36      for( j=1; j<N+2; j++ )
37      {
38          grid[0][j] = 1.0;
39          new_grid[0][j] = 1.0;
40      }
41  }
42
43  int main(void)
44  {
45      struct timeval ts_st, ts_end;
46      double dTmax, dT, epsilon, max_time;
47      int finished, i, j, k, l;
48      double T;
49      int nr_iter;
50
51      if( MYTHREAD == 0 )
52          initialize();
53
54      /* ------------------------------------------------------------------ */
55      /* The next for loop is to initialize the pointers ptr[] and
56         ptr_new[] */
57      for( i=0; i<N+2; i++ )
58      {
59          ptr[i] = &grid[i][0];
60          new_ptr[i] = &new_grid[i][0];
61      }
62      /* ------------------------------------------------------------------ */
```

```
63
64      epsilon  = 0.0001;
65      finished = 0;
66      nr_iter = 0;
67
68      upc_barrier;
69
70      gettimeofday( &ts_st, NULL );
71
72      do
73      {
74          dTmax = 0.0;
75          upc_forall( i=1; i<N+1; i++; i*THREADS/(N+2) )
76          {
77              for( j=1; j<N+1; j++ )
78              {
79                  /* ------------------------------------------------------ */
80                  /* this section has to be changed, to use ptr[][]
81                     and new_ptr[][] */
82                  T = 0.25 *
83                      ((*ptr[i+1][j]) + (*ptr[i-1][j]) +
84                       (*ptr[i][j-1]) + (*ptr[i][j+1])); /* stencil */
85                  dT = T - ptr[i][j];
86                  new_ptr[i][j] = T;
87                  /* ------------------------------------------------------ */
88                  if( dTmax < fabs(dT) )
89                      dTmax = fabs(dT);
90              }
91          }
```

```
92
93          dTmax_local[MYTHREAD] = dTmax;
94          upc_barrier;
95          dTmax = dTmax_local[0];
96          for( i=1; i<THREADS; i++ )
97              if( dTmax < dTmax_local[i] )
98                  dTmax = dTmax_local[i];
99          /* ------------------------------------------------------------ */
100         /* here a barrier as to be introduced to avoid race
101            conditions over dTmax_local[i] */
102         upc_barrier;
103
104         if( dTmax < epsilon )
105             finished = 1;
106         else
107         {
108             /* ------------------------------------------------------------ */
109             /* here swap the pointers ptr[] and new_ptr[] thru tmp_ptr
110 Note: a shared pivot pointer has to be declared */
111             for( k=0; k<N+2; k++ )
112             {
113                 tmp_ptr    = ptr[k];
114                 ptr[k]     = new_ptr[k];
115                 new_ptr[k] = tmp_ptr;
116             }
117             /* ------------------------------------------------------------ */
```

```
118
119             /* the barrier can be removed here since each thread has its
120                own private pointers to grid[][] and new_grid[][] whereas in
121                the previous case we wanted to wait till copying
122                new_grid[][] to grid[][] is completed by all threads */
123         }
124         nr_iter++;
125     } while( finished == 0 );
126
127     gettimeofday( &ts_end, NULL );
128
129     exectime[MYTHREAD] = ts_end.tv_sec + (ts_end.tv_usec / 1000000.0);
130     exectime[MYTHREAD] -= ts_st.tv_sec + (ts_st.tv_usec / 1000000.0);
131
132     upc_barrier;
133
134     if( MYTHREAD == 0 )
135     {
136         max_time = exectime[MYTHREAD];
137         for( i=1; i<THREADS; i++ )
138             if( max_time < exectime[i] )
139                 max_time = exectime[i];
140         printf("%d iterations in %.3lf sec\n", nr_iter, max_time);
141     }
142
143     return 0;
144 }
145
```

4.6 - Performance boost using privatization

In the new version I improved the program performance by using privatization. Indeed, it has been shown that the speed of UPC memory accesses greatly differs on the type of access performed. The private memory access is the fastest. In the following program the different threads are working with new private pointers which are *ptr_priv[] and *new_ptr_priv[]. Their size is only the number of local rows which contain the data that corresponding to each threads, this is not the total number of rows.

To do this implementation I needed to adapt the previous program. I splitted the original upc_forall(...) loop into three blocks (l. 60-100) to separate the processing of the first row, middle part and last row. We need to split this loop because now threads are only working with private pointers which are not pointing to the globals arrays. In other words, each thread can only accesses to a part of the global data.

This new optimization needed too the adaptation of other part of the program, as the pointer flipping (l. 117) for example.

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

#define N 30


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

shared [N+2] double *ptr[N+2], *new_ptr[N+2], *tmp_ptr;
/* New private pointers */
double *ptr_priv[N+2], *new_ptr_priv[N+2], *tmp_ptr_priv;

void initialize(void)
{
    int j;

    for( j=1; j<N+2; 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, max_time;
    int finished, i, j, k, l;
    double T;
```

```
34      int nr_iter;
35
36      if( MYTHREAD == 0 )
37          initialize();
38
39
40      for( i=0; i<N+2; i++ )
41      {
42          ptr[i] = &grid[i][0];
43          new_ptr[i] = &new_grid[i][0];
44      }
45      /* to initialize the private pointers */
46      ptr_priv = &grid[MYTHREAD][0];
47      new_ptr_priv = &new_grid[MYTHREAD][0];
48
49      epsilon  = 0.0001;
50      finished = 0;
51      nr_iter = 0;
52
53      upc_barrier;
54
55      gettimeofday( &ts_st, NULL );
```

```
56
57      do
58      {
59          dTmax = 0.0;
60          /* block 1 */
61          upc_forall( i=0; i<1; i++; i*THREADS/(N+2) )
62          {
63              for( j=0; j<1; j++ )
64              {
65                  T = 0.5 * ((*ptr_priv[i+1][j]) + (*ptr_priv[i][j+1])); /* stencil */
66                  dT = T - ptr_priv[i][j];
67                  new_ptr_priv[i][j] = T;
68
69                  if( dTmax < fabs(dT) )
70                      dTmax = fabs(dT);
71              }
72          }
73          /* block 2 */
74          upc_forall( i=1; i<N+1; i++; i*THREADS/(N+2) )
75          {
76              for( j=1; j<N+1; j++ )
77              {
78                  T = 0.25 *
79                      ((*ptr_priv[i+1][j]) + (*ptr_priv[i-1][j]) +
80                       (*ptr_priv[i][j-1]) + (*ptr_priv[i][j+1])); /* stencil */
81                  dT = T - ptr_priv[i][j];
82                  new_ptr_priv[i][j] = T;
83
84                  if( dTmax < fabs(dT) )
85                      dTmax = fabs(dT);
86              }
87          }
```

```
88          /* block 3 */
89          upc_forall( i=N; i<N+2; i++; i*THREADS/(N+2) )
90          {
91              for( j=N; j<N+2; j++ )
92              {
93                  T = 0.5 * ((*ptr_priv[i-1][j]) + (*ptr_priv[i][j-1])); /* stencil */
94                  dT = T - ptr_priv[i][j];
95                  new_ptr_priv[i][j] = T;
96
97                  if( dTmax < fabs(dT) )
98                      dTmax = fabs(dT);
99              }
100         }
101
102         dTmax_local[MYTHREAD] = dTmax;
103         upc_barrier;
104         dTmax = dTmax_local[0];
105         for( i=1; i<THREADS; i++ )
106             if( dTmax < dTmax_local[i] )
107                 dTmax = dTmax_local[i];
108
109         upc_barrier;
```

```
110
111         if( dTmax < epsilon )
112             finished = 1;
113         else
114         {
115             for( k=0; k<N+2; k++ )
116             {
117                 /* Pointer fliping for private pointers */
118                 tmp_ptr_priv     = ptr_priv[k];
119                 ptr_priv[k]      = new_ptr_priv[k];
120                 new_ptr_priv[k] = tmp_ptr_priv;
121                 /* --- */
122                 tmp_ptr     = ptr[k];
123                 ptr[k]      = new_ptr[k];
124                 new_ptr[k] = tmp_ptr;
125             }
126         }
127         nr_iter++;
128     } while( finished == 0 );
129
130     gettimeofday( &ts_end, NULL );
131
132     exectime[MYTHREAD] = ts_end.tv_sec + (ts_end.tv_usec / 1000000.0);
133     exectime[MYTHREAD] -= ts_st.tv_sec + (ts_st.tv_usec / 1000000.0);
134
135     upc_barrier;
136
137     if( MYTHREAD == 0 )
138     {
139         max_time = exectime[MYTHREAD];
140         for( i=1; i<THREADS; i++ )
141             if( max_time < exectime[i] )
142                 max_time = exectime[i];
143         printf("%d iterations in %.3lf sec\n", nr_iter, max_time);
144     }
145
146     return 0;
147 }
148
149
```

4.7 - Dynamic problem size

In this last version, I implemented the dynamic allocating shared memory during execution to make the problem size dynamic. In other word, user can specifying N at runtime. As we can see in the program, now all pointer are declare without any dependencies to N.

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

//#define N 30
int N;

shared [] double grid[][], new_grid[][];
shared double exectime[THREADS];
shared double dTmax_local[THREADS];

shared [] double *ptr[], *new_ptr[], *tmp_ptr;
/* New private pointers */
double *ptr_priv[], *new_ptr_priv[], *tmp_ptr_priv;

void initialize(void)
{
    int j;

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

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

```
34      int nr_iter;
35
36      if(argc != 2) {
37          if(MYTHREAD == 0)
38              printf("Erreur : Un parametre est attendu. %s\n", argv[0]);
39          return -1;
40      }
41
42      /* to get N */
43      if(argv[1] <= 0) {
44          if(MYTHREAD == 0)
45              printf("Erreur : N should be bigger than 0 \n");
46          return -2;
47      }
48      N = argv[1];
49
50      /* dynamic allocation */
51      if( MYTHREAD == 0 ) {
52          // grid & new_grid
53          grid = (shared[N+2] double *) upc_alloc((N+2)*(N+2) * sizeof(double));
54          new_grid = (shared[N+2] double *) upc_alloc((N+2)*(N+2) * sizeof(double));
55          // ptr & new_ptr
56          ptr = (shared[] double *) upc_alloc((N+2) * sizeof(double));
57          new_ptr = (shared[] double *) upc_alloc((N+2) * sizeof(double));
58      }
59      upc_barrier;
60      // ptr_priv & new_ptr_priv
61      ptr_priv = (shared[] double *) upc_alloc((N+2) * sizeof(double));
62      new_ptr_priv = (shared[] double *) upc_alloc((N+2) * sizeof(double));
63      upc_barrier;
64
```

```
65
66      if( MYTHREAD == 0 )
67          initialize();
68
69      for( i=0; i<N+2; i++ )
70      {
71          ptr[i] = &grid[i][0];
72          new_ptr[i] = &new_grid[i][0];
73      }
74      /* to initialize the private pointers */
75      ptr_priv = &grid[MYTHREAD][0];
76      new_ptr_priv = &new_grid[MYTHREAD][0];
77
78      epsilon  = 0.0001;
79      finished = 0;
80      nr_iter = 0;
81
82      upc_barrier;
83
84      gettimeofday( &ts_st, NULL );
85
86      do
87      {
88          dTmax = 0.0;
89          /* block 1 */
90          upc_forall( i=0; i<1; i++; i*THREADS/(N+2) )
91          {
92              for( j=0; j<1; j++ )
93              {
94                  T = 0.5 * ((*ptr_priv[i+1][j]) + (*ptr_priv[i][j+1])); /* stencil */
95                  dT = T - ptr_priv[i][j];
96                  new_ptr_priv[i][j] = T;
```

```
97
98                  if( dTmax < fabs(dT) )
99                      dTmax = fabs(dT);
100             }
101         }
102         /* block 2 */
103         upc_forall( i=1; i<N+1; i++; i*THREADS/(N+2) )
104         {
105             for( j=1; j<N+1; j++ )
106             {
107                 T = 0.25 *
108                     ((*ptr_priv[i+1][j]) + (*ptr_priv[i-1][j]) +
109                      (*ptr_priv[i][j-1]) + (*ptr_priv[i][j+1])); /* stencil */
110                 dT = T - ptr_priv[i][j];
111                 new_ptr_priv[i][j] = T;
112
113                 if( dTmax < fabs(dT) )
114                     dTmax = fabs(dT);
115             }
116         }
117         /* block 3 */
118         upc_forall( i=N; i<N+2; i++; i*THREADS/(N+2) )
119         {
120             for( j=N; j<N+2; j++ )
121             {
122                 T = 0.5 * ((*ptr_priv[i-1][j]) + (*ptr_priv[i][j-1])); /* stencil */
123                 dT = T - ptr_priv[i][j];
124                 new_ptr_priv[i][j] = T;
125
126                 if( dTmax < fabs(dT) )
127                     dTmax = fabs(dT);
128             }
129         }
```

```
130
131         dTmax_local[MYTHREAD] = dTmax;
132         upc_barrier;
133         dTmax = dTmax_local[0];
134         for( i=1; i<THREADS; i++ )
135             if( dTmax < dTmax_local[i] )
136                 dTmax = dTmax_local[i];
137
138         upc_barrier;
139
140         if( dTmax < epsilon )
141             finished = 1;
142         else
143         {
144             for( k=0; k<N+2; k++ )
145             {
146                 /* Pointer fliping for private pointers */
147                 tmp_ptr_priv     = ptr_priv[k];
148                 ptr_priv[k]      = new_ptr_priv[k];
149                 new_ptr_priv[k] = tmp_ptr_priv;
150                 /* --- */
151                 tmp_ptr     = ptr[k];
152                 ptr[k]      = new_ptr[k];
153                 new_ptr[k] = tmp_ptr;
154             }
155         }
156         nr_iter++;
157     } while( finished == 0 );
158
159     gettimeofday( &ts_end, NULL );
```

```
160
161     exectime[MYTHREAD] = ts_end.tv_sec + (ts_end.tv_usec / 1000000.0);
162     exectime[MYTHREAD] -= ts_st.tv_sec + (ts_st.tv_usec / 1000000.0);
163
164     upc_barrier;
165
166     if( MYTHREAD == 0 )
167     {
168         max_time = exectime[MYTHREAD];
169         for( i=1; i<THREADS; i++ )
170             if( max_time < exectime[i] )
171                 max_time = exectime[i];
172         printf("%d iterations in %.3lf sec\n", nr_iter, max_time);
173     }
174
175     return 0;
176 }
177
178
```