Parallel Programming. OpenMP

Bruno Cuevas, Albert Cuevas, Iñigo Morcillo

19th October 2016

1 Introduction

In the previous task proposed within this subject, a C binary was compiled which was able to:

- Obtain an answer to the proposed problem (vibration of a string using finite differences).
- Use dynamical memory allocation.
- Improve the performance of the binary through reducing the amount of memory necessary to arrive to the final answer.

In this task, it has been proposed to make a parallel program through the use of shared memory paradigm, implemented for C in OpenMP.

2 Code

The program has been built using parallel sections in the iteration over the elements of the array for each time step. Since the program does only store in memory the lines that are being employed for the current calculation, then it is necessary to realocate the elements, task that is also performed in parallel.

Since parallel operations are in 'no wait', then a barrier had to be introduced between the two parallel loops of the section in order to avoid the realocation of elements that have not been processed yet, which would lead to undeterminism (we could obtain different results in each run) and error.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "omp.h"
int main(int argc, char *argv[]) {
        int X = atoi(argv[1]);
        int T = atoi(argv[2]);
        // oS : operation-matrix Size
```

```
                int oS = 3;
                double *U = malloc(sizeof(double)*(X+1)*3);
                int t, x;
                double L= 0.345678, S;
                // initialize positions of matrix U
                for (x=1; x<X; x++) {
                        U[x] = sin(x*M_PI/(double)X);
                        U[x + X + 1] = U[x] * cos(M_PI
↪  /(double)T);
                }
                for (t=0; t<oS; t++) {
                        U[t*(X+1)] = U[t*(X+1) + X] = 0.0;
                }
                // Program Body
                        // tS : timeStep
                        // iX : iterX
                        // fstTerm : First term of the equation
                        // sndTerm : Second term of the equation
                        // trdTerm : Third term of the equation
                        // fthTerm : Fourth term of the equation
                        // iRXL : index X for relocation
                        // iRTL : index T for relocation
                int tS, iX, address, iRXL;
                double fstTerm, sndTerm, trdTerm, fthTerm;



                for (tS = 1; tS < T + 2; tS ++) {
                        #pragma omp parallel default(none)
↪  shared(tS, T, U, L, X) private(address, fstTerm, sndTerm,
↪  trdTerm, fthTerm, iX, iRXL)
                        {
                        #pragma omp for nowait
                        for (iX = 1; iX < X ; iX ++) {
                        address = ((X+1)) + iX;
                        fstTerm = 2*(1-L)*U[address];
                        sndTerm = L*U[address + 1];
                        trdTerm = L*U[address - 1];
                        fthTerm = -U[address - (X+1)];
                        U[address + (X+1)] = fstTerm + sndTerm +
↪  trdTerm + fthTerm ;

                        }
                        #pragma omp barrier
                        #pragma omp for nowait
                        for (iRXL = 1 ; iRXL < X ; iRXL ++) {
```

```
                U[iRXL] = U[iRXL + X + 1];
                U[iRXL + X + 1] = U[iRXL + 2*(X+1)];
                }
                }
        }


        // obtain checksum of final state
                // cS : checkSum for a given value
        S = 0;
        int fP = (X+1);
        for (iX = 0; iX <= X; iX ++){
                S = S + U[fP + iX];
        }
        printf("CheckSum = %e\n", S);
        return 0;
}
```

To run the code, use:

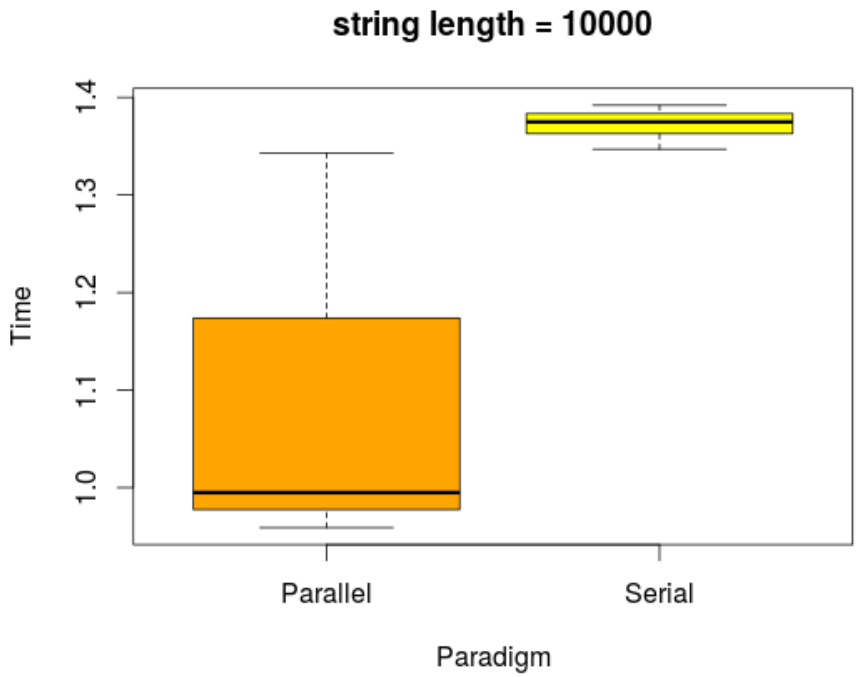
```
$ ./e1OMP LENGTH TIME
```

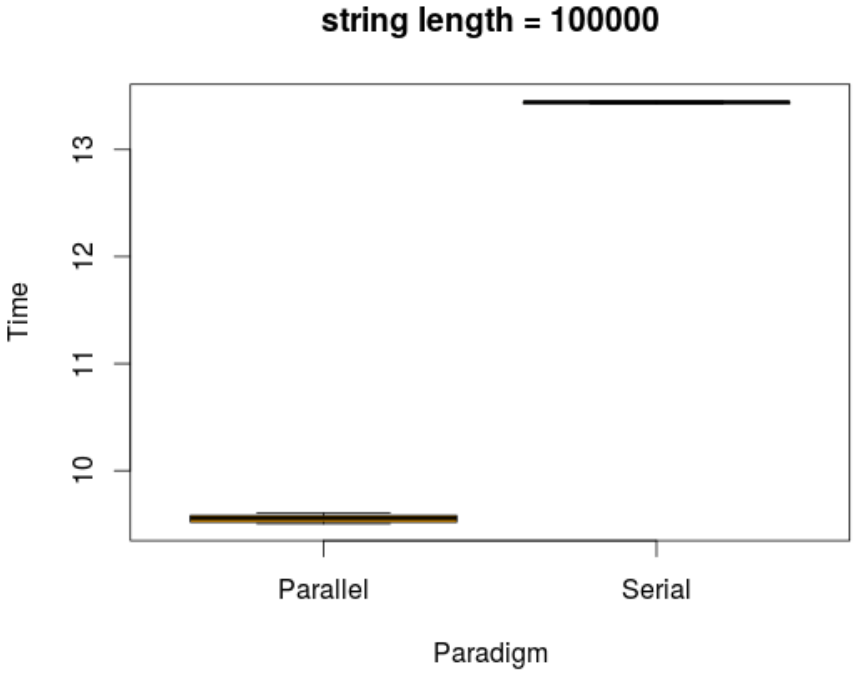
To compile the code, use:

```
$ gcc -Wall ejercicioOMP.c -o e1OMP -lm -fopenmp
```

3 Performance

The program has been tested in a 4 threads computer against its predecessor, the non-parallel binary, using the shell tool -temp-. Real execution-time was tested for two array sizes, 10000 and 100000.


string length = 10000
Time
1.0
1.1
1.2
1.3
1.4
Parallel
Serial
Paradigm


string length = 100000
Time
10
11
12
13
Parallel
Serial
Paradigm

In both cases, a t-test was done to test if differences were significative. The p-value for the 10000 length array experiment was 0.0026, and for the 100000 this was much lower (10^{-15}), so we can conclude that the parallel implementation has a different speed (in this case, a larger speed) than the not-parallel implementation.

Since the computer where it has been tested has 4 cores, we would have expected a 4 times speed-up. However, it does only run 40% faster.

A hypothesis for this speed-up observed is that the process of opening and clossing threads in each time step leads to an important loss of speed. Another source of loss of speed in our implementation is the reallocation of terms in the matrix. However, it helps us saving memory, so we consider that it is not actually a loss of efficiency, only of speed.