TP2 Parallel Programming

Notes

The goal of this work is to get started with OMP Note that `#include <omp.h>` must be used to work with it I am also using `std::chrono::high_resolution_clock` for time precision.

Here I used `omp_set_num_threads` to manually set the amount of threads we use from the CPU. **It is good to know that in every exercice, if we use more threads than we have in our CPU, it worthens the results** Now this is noted here and I won't come back to it after in the exercices

MoyenneVecteurs

Each function initializes vectors and time.

scalar

```
for (unsigned long int i = 0; i < size; i++) {
    result[i] = (vector1[i] + vector2[i])/2;
}
```

This is the same work as in the first TP

vectorial

```
omp_set_num_threads(n_threads);
#pragma omp parallel for schedule(static)
    for (unsigned long int i = 0; i < size; i++) {
        S[i] = (A[i] + B[i]) / 2;
    }
```

We already talked about setting the amount of threads. Here `#pragma omp parallel for` means that the next for loop will be parallelized over the threads we decided to use. Here we use `schedule(static)` and we will cover that on exercice 3. Schedule is just a way to decide how we attribute the elements to sum over the threads.

Two Vectors Scalar Product

Here we need to multiply element by element and then sum.

scalar

```
for (unsigned long int i = 0; i < size; i++) {
    scalarProduct += (vector1[i] * vector2[i])/2;
```

```
}
```

vectorial

```
float s = 0;
omp_set_num_threads(n_threads);
#pragma omp parallel
{
    float local_result=0;
    #pragma omp for schedule(static)
    for (unsigned long int i = 0; i < size; i++) {
        local_result += A[i] * B[i];
    }
    #pragma omp critical
        s += local_result;
}
return s;
```

The first lines are the same thing as the exercise before. We are just multiplying bit by bit instead of adding.

This may be the same thing but there is one difference: `result` is being shared by all the threads for every addition. Therefore parallelizing doesn't do a thing. Here we rewrite it to use a `local_result` for every thread, and use `pragma omp critical` to show that we just need to add the `local_results` from every threads.

Scalar Shafted

To mess around with the type of scheduling, we had to make sure all calculations would take the same time. So we chose a particular seed for the creation of random vectors, which means it was always the same vector.

Auto

`auto` simply choses how to distribute the calculations to the threads for us.

Séquentiel	Parallel 1	Parallel 2	Parallel 4	Parallel 8
6.1e-8	3.6e-10	2.0e-10	1.3e-10	1.3e-10

Static, size

Note: SIZE is the size of the vector

The static mode devide the for loop calculations into chunks of fixed size. This means that if we specify `size = 1000`, the first 1000 iterations are done by thread 1, 1001 to 2000 by thread 2... And then it loops.

We can therefore specify a nicely chosen chunk size (probably something that devides the total amount of iterations). Here are some tests.

No size given

Séquentiel	Parallel 1	Parallel 2	Parallel 4	Parallel 8
6.1e-8	3.59e-10	2.0e-10	1.3e-10	1.29e-10

This is the same results as `auto`, because the compileur knows what chunk size is best, and `static` is the mode out of the available modes.

SIZE/2

Séquentiel	Parallel 1	Parallel 2	Parallel 4	Parallel 8
6.1e-8	3.59e-10	2.0e-10	2.0e-10	2.28e-10

We have the same results as before for chunk sizes that let `amount of chunks < amount of threads`

SIZE/4

Séquentiel	Parallel 1	Parallel 2	Parallel 4	Parallel 8
6.1e-8	3.6e-10	2.0e-10	1.3e-10	1.45e-10

Same remark

Size: 1000

Séquentiel	Parallel 1	Parallel 2	Parallel 4	Parallel 8
6.1e-8	6.2e-10	3.1e-10	1.55e-10	1.69e-10

Lots of chunks this time, which is very bad when we limit the amount of threads, but quite good on 4 threads.

Dynamic, size

Dynamic scheduling gives each threads a bloc of `size` chunks, and then gives the next bloc to the first thread that frees itself, and so on.

Séquentiel	Parallel 1	Parallel 2	Parallel 4	Parallel 8
6.1e-8	9.3e-9	1.7e-8	3.4e-8	3.0e-8

The results here are very bad, because default `size` is 1, meaning that we dynamically attribute the calculations one at a time.

I won't add all the results, but it is worth noting that `Dynamic, size/4` is not as good as `Static, size/4` even though it sounds like it is the same scheduling.

Guided, size

This is the same as dynamic except that the size of the chunks decreases exponentially, with at least `size` iterations in a chunk.

Gaussian Filter

Sequential

```
for (auto i = 1; i < height - 1; i++)
{
    for (auto j = 1; j < width - 1; j++)
    {
        if ((i == 0) || (i == height - 1) || (j == 0) || (j == width - 1))
        {
            Resultat[i][j] = 0;
        }
        else
        {
            Resultat[i][j] = std::abs(Source[i - 1][j - 1] + Source[i - 1]
[j] + Source[i - 1][j + 1] - (Source[i + 1][j - 1] + Source[i + 1][j] +
Source[i + 1][j + 1]));
            Resultat[i][j] += std::abs(Source[i - 1][j - 1] + Source[i][j
- 1] + Source[i + 1][j - 1] - (Source[i - 1][j + 1] + Source[i][j + 1] +
Source[i + 1][j + 1]));
        }
    }
}
```

Gaussian Filter is about using element before and element after, so I first need to deal with side effects.

Vectorial

```
#pragma omp parallel for
for (auto i = 1; i < height - 1; i++)
{
    for (auto j = 1; j < width - 1; j++)
    {
        if ((i == 0) || (i == height - 1) || (j == 0) || (j == width - 1))
        {
            Resultat[i][j] = 0;
        }
        else
        {
            Resultat[i][j] = std::abs(Source[i - 1][j - 1] + Source[i - 1]
[j] + Source[i - 1][j + 1] - (Source[i + 1][j - 1] + Source[i + 1][j] +
Source[i + 1][j + 1]));
            Resultat[i][j] += std::abs(Source[i - 1][j - 1] + Source[i][j
- 1] + Source[i + 1][j - 1] - (Source[i - 1][j + 1] + Source[i][j + 1] +
Source[i + 1][j + 1]));
        }
    }
}
```

I did some tests trying to parallelized both for loops, or to look into nested parallelism, but it seems fitting that parallelizing line by line is the best way to do it.

List Creation

Sequential

```
int sequentiel(int* M, int* S, unsigned long int size) {
    unsigned long int ne = 0;
    for (unsigned long int i = 0; i < size; i++) {
        if ( M[i] % 2 == 0 ) {
            S[ne] = M[i];
            ne += 1;
        }
    }
    return ne;
}
```

Vectorial

```
omp_set_num_threads(n_threads);
unsigned long int ne = 0;
int *T;
T = (int *) malloc(size * sizeof(int));
for (auto i = 0; i < size; i++) {
    T[i] = 200;
}


#pragma omp parallel shared(ne)
{
    #pragma omp for reduction(+:ne) schedule(static)
    for (unsigned long int i = 0; i < size; i++) {
        if ( M[i] % 2 == 0 ) {
            T[i] = M[i];
            ne += 1;
        }
    }
}


int ind = 0;
for (auto i = 0; i < size; i++) {
    if (T[i] != 200) {
        S[ind] = T[i];
        ind += 1;
    }
}
```

```
  free(T);

  return ne;
```

This kind of test. Trying with a critical counter and list, trying with a reduction for the counter, etc... I just need to know that I must be careful with shared resources.