CSC5001

Lab 1 : Introduction to OpenMP

1 Initialization of a team of threads

The directive `#pragma omp parallel` indicates that the statement or block of instructions following this directive must be executed by a *team* of threads. The statement or block affected by this directive is called *parallel section*.

```
1 int main()
2 {
3
4   printf("Hello !\n");
5   printf("Bye !\n");
6
7   return 0;
8 }
```

1. Modify the `hello.c` program by adding a directive `#pragma omp parallel` to the line 3.
2. Compile and run the program as follows :

   ```
   gcc -o hello hello.c
   ./hello
   ```

3. Recompile using the `-fopenmp` option to `gcc` this time. Run it several times.How many threads are used ? Compare this number with the number of logical cores of your computer. When is printed the message `Bye!` ?
4. The `OMP_NUM_THREADS` environment variable allows to manage the number of threads to be used by default. Run the program thanks to the command :

   ```
   OMP_NUM_THREADS=13 ./hello
   ```

5. The `omp_get_thread_num` function returns the identifier of the calling thread. It requires adding the OpenMP header file : `#include <omp.h>`. Modify the two calls to `printf` in order to also display the calling thread identifier. Compile and run several times the program by varying the threads number. Does the scheduling remain the same ?
6. Let also print the message `Bye !` by each threads by including both printings within a single parallel section. Run with a dozen of threads.
7. The `#pragma omp barrier` directive allows to a team of threads to wait for each other at a precise line in the program. Insert this directive between the two `printf` calls. Compile and run the program.

2 Critical section

The `#pragma omp critical` directive implies that each thread executes alone the statement or the statement block that follows. The code protected by the `critical` directive is executed in *mutual exclusion*. Use this directive in order to avoid interleaving printing (note : remove the barrier added at the previous exercice).

3 Shared vs private variable

In the `sharing.c` program, `k` and `i` variables are declared in the `main` function.

```
1  int main()
2  {
3    int k = 0;
4
5    #pragma omp parallel
6    {
7      int i;
8      for(i = 0; i < 100000; i++)
9          k++;
10   }
11
12   printf("nbthreads x 100000 = %d\n",k);
13   return 0;
14 }
```

The variable `k` is declared before the call to the `parallel` directive, this variable will be shared between threads : any thread in the next parallel section will be able to consult/modify this variable at will.

The variable `i` is declared inside the parallel section and is private to the thread : each thread will have its own copy and will not be able to modify the others.

1. Compile and run the `sharing.c` program. You can observe that the value of `k` is sometimes equal to 100 000 times the number of used threads, sometimes the `k++` incrementations does not seem to be executed and the final value of `k` changes for an un to another. *Manipulate a shared variable without any care leads to an incoherent result.*
2. Insert a critical section in line 9 so as not to lose any incrementation. Compile and test.

4 For loop distribution

We aim to parallelize the following code where the calls to `compute(i)` are known as independant.

```
1 for(int i = 0; i < N; i++)
2    compute(i);
```

For that purpose, we will distribute the loop iterations among the different threads making them computed only one time. The following OpenMP code automatically makes it :

```
#pragma omp parallel
#pragma omp for schedule(static)
  for(int i = 0; i < N; i++)
     compute(i);
```

4.1 Loop index distribution policies

The `#pragma omp for` directive indicates that the loop indexes have to be distributed among the threads and the `schedule` clause precises the distribution policy to use. Four different main policies are available :

- `schedule(static)` divides equally indexes by the number of threads and distributes these blocks of consecutives indexes to the threads;.
- `schedule(static, k)` cyclically distributes iterations by blocks of k indexes;
- `schedule(dynamic, k)` distributes iterations on demand by blocks of k indexes;
- `schedule(guided, k)` for those who are curious.

Note that the `#pragma omp parallel for` directive is an all together sequence of directives `#pragma omp parallel` and `#pragma omp for`.

Study the different policies thanks to the `for-loop.c` program :

```
1 int main()
2 {
3   int i;
4
5   for(i=0; i < 40; i++)
6     printf("%d computes %i\n",omp_get_thread_num(),i);
7
8   return 0;
9 }
```

1. After adding the `#pragma omp parallel for` directive in line 4, compile and run the program with 4 threads several times.
2. Modify the program with different distributon policies. Use the fommowing command :

   ```
   OMP_NUM_THREADS=4 ./boucle-for | sort -n -k 3
   ```

   to visualize easily the work of each thread.

4.2 Computation of the sum of a table - reduction

We have seen that it is necessary to protect access to shared variables. This can be done using critical sections but also using atomic instructions. An atomic section is a critical section reduced to a single instruction. We will compare the cost of different protection techniques on the following code :

```
1 for (i=0; i < N; i++)
2   sum += tab[i];
```

Replace the `TODO` in the `sum.c` program with the following techniques :

1. section `#pragma omp parallel for` where the shared variable `sum` is accessed in a critical section ;
2. section `#pragma omp parallel for` where the shared varaibel `sum`is accessed in a atomic section `#pragma omp atomic` ;
3. use of a local variable `my_sum` in which each thread accumulates values on its own during the loop, then adds its contribution to `sum` all at once ;
4. reduction `#pragma omp parallel for reduction(+:sum)` : the compiler transforms the program to introduce a local variable in order to minimize the time spent in the critical section.

5 EasyPAP

We are now going to look at programs that manipulate images, which are 2 dimensional arrays containing pixels.

To facilitate development, we're going to use an environment called EASYPAP, which allows you to display images, launch calculations and interactively visualize image changes at each iteration.

To retrieve EASYPAP, do :

```
git clone https://gitlabense.imtbs-tsp.eu/elisabeth.brunet/csc5001-easypap-se.git
cd csc5001-easypap-se
cd lib/ezv
make
cd ..
make
```

Documentation is available here : https://gforgeron.gitlab.io/easypap/

If successfull, you should observe an animated image when taping :

```
./run -k spin
```

5.1 Getting started

The `kernel/c/spin.c` file contains several variants of the `spin` kernel. Let begin by modifying (temporalty) the sequential variant in order to understand how the image is browsed.

Apply the next modification (Figure 1, line 7) in the code of `spin_compute_seq` function.

Recompile, run the kernel and verify the effect of the modification :

```
./run --kernel spin --variant seq
```

```
1  unsigned spin_compute_seq (unsigned nb_iter)
2  {
3    for (unsigned it = 1; it <= nb_iter; it++) {
4
5      for (int i = 0; i < DIM; i++)
6        for (int j = 0; j < DIM; j++)
7          if (j < DIM/2)  // Try also: if (i < DIM/2 || j < DIM/2)
8            cur_img (i, j) = compute_color (i, j);
9
10     rotate (); // Slightly increase the base angle
11   }
12
13   return 0;
14 }
```

FIGURE 1 – Basic sequential version of `spin` kernel (`kernel/c/spin.c`).

```
1  unsigned spin_compute_tiled (unsigned nb_iter)
2  {
3    for (unsigned it = 1; it <= nb_iter; it++) {
4
5      for (int y = 0; y < DIM; y += TILE_H)
6        for (int x = 0; x < DIM; x += TILE_W)
7          if ((x / TILE_W + y / TILE_H) % 2)
8            do_tile (x, y, TILE_W, TILE_H);
9
10     rotate ();
11   }
12
13   return 0;
14 }
```

FIGURE 2 – Tiled version of `spin` kernel.

Now let's study the `tiled` variant (Figure 2), that virtually subdivides the image in tiles of size `TILE_W` × `TILE_H` pixels.

The `do_tile (int x, int y, int w, int h)` function takes in charge the computation of pixels in the rectangle defined by the parameters. This function is basically a switch whose default branch is the `spin_do_tile_default` function.

Apply the modification shown in Figure 2, ligne 7 in the code of the `spin_compute_tiled` function. Recompile and verifiy the effect on the display :

```
./run --kernel spin --variant tiled
```

Vary the tile size choosing values in power of two thanks to the `--tile-size` (or `-ts`) option For instance, launch :

```
./run --kernel spin --variant tiled --tile-size 8
```

You can also define rectangular tiles with options `--tile-width` et `tile-height` :

```
./run --kernel spin --variant tiled --tile-width 64 --tile-height 8
```

The option `--monitoring` (or `-m`) opens *monitoring* windows that show the activity of each threads and the area on which they compute :

```
./run --kernel spin --variant tiled --tile-width 64 --tile-height 8 --monitoring
```

Take a moment to contemplate... then remove the line_7.

5.2 First step with OpenMP

The next step is to write parallel versions of the kernel `spin` using OpenMP.

To do this, copy the function `spin_compute_tiled` to a new variant called `spin_compute_omp`.

```
1  unsigned spin_compute_omp (unsigned nb_iter)
2  {
3    for (unsigned it = 1; it <= nb_iter; it++) {
4
5  #pragma omp parallel for
6      for (int y = 0; y < DIM; y += TILE_H)
7        for (int x = 0; x < DIM; x += TILE_W)
8          do_tile (x, y, TILE_W, TILE_H);
9
10     rotate ();
11   }
12
13   return 0;
14 }
```

FIGURE 3 – Basic OpenMP version of `spin` kernel.

Apply the modifications shown in Figure 3, recompile and test your newly parallel version :

```
./run -k spin -v omp -m
```

What do you observe ? Try different tile size.

You can also modify the *threads* number used by OpenMP thanks to the `OMP_NUM_THREADS` environment variable. For instance :

```
OMP_NUM_THREADS=12 ./run -k spin -v omp -m -ts 4
```

5.3 Work distribution visualization

You are now in a position to experiment different for loop index scheduling strategies by observing the distribution of pixels per CPU on the tile *monitoring* window.

For example, you can add the clause `schedule (static,1)`, recompile, and view the change by restarting :

```
./run -k spin -v omp -m
```

To play with all scheduling strategies without necessarily recompile the code between each run, you can use the `schedule (runtime)` inline clause 5, which allows you to specify the strategy to be used via the environment variable `OMP_SCHEDULE` environment variable. So, once the code has been recompiled, you can run sequentially :

```
OMP_SCHEDULE=static ./run -k spin -v omp -m -ts 1

OMP_SCHEDULE=static,1 ./run -k spin -v omp -m -ts 1

OMP_SCHEDULE=dynamic ./run -k spin -v omp -m -ts 1
```

5.4 Speedup

To determine the speedup achieved, i.e. the gain obtained by from a sequential to a multithreaded version, we first precisely measure the time obtained by each version by disabling display (option `--no-display` or `-n`). For example :

```
[my-machine] ./run -k spin -v seq -i 200 -n
Using kernel [spin], variant [seq]
Computation completed after 200 iterations
6203.184

[my-machine] ./run -k spin -v omp -i 200 -n
Using kernel [spin], variant [omp]
Computation completed after 200 iterations
841.775
```

In this example, the speedup is $\frac{6203}{841} \approx 7.375$. Calculate your own.

5.5 Column parallelization

Move your `#pragma omp parallel for` just before the `for (int x)` loop and recompile. Observe that the work repartition is now done by column :

```
OMP_SCHEDULE=static ./run -k spin -v omp -m -ts 1
```

To find out whether this version is more efficient than parallelization by line, perform a comparable time measurement (in number of iterations) with previous experiments.

For a fair comparison, put a `#pragma omp parallel` before the loop `for (int y)` and use only `#pragma omp for` in front of the loop `for (int x)` loop. For instance, test :

```
OMP_SCHEDULE=static ./run -k spin -v omp -i 200 -n -ts 1
```

Do the results vary if a cyclic distribution is used ? Test :

```
OMP_SCHEDULE=static,1 ./run -k spin -v omp -i 200 -n -ts 1
```

5.6 Collapse

Duplicate the `spin_compute_omp` function to create a function `spin_compute_omp_tiled`. Modify the code by adding the clause `collapse(2)` to the OpenMP **`for`** directive.

The `collapse(n)` clause allows to distribute n-tuples of indices rather than one-dimensional indexes by collapsing `n` nested loops.

Observe the influence of this clause on tile distribution by varying the the distribution policy.