TP - Introduction to OpenMP

Oguz Kaya
oguz.kaya@universite-paris-saclay.fr

To compile the program `program.cpp` with OpenMP and generate the executable `program`, type the following command in the terminal:

```
g++ -O2 -std=c++11 -fopenmp program.cpp -o program
```

Part 1

Hello World using OpenMP

Ex. 1

a) Write a program `hello-openmp.cpp` (skeleton code given) having a parallel region in which each thread prints its identifier as well as the total number of threads.

b) Next, in the same parallel region, print "Hello World from threadId=??" by a single thread with its thread id. Try using both `omp single` and `omp master` constructs and observe the difference.

c) Try to modify the number of threads in the parallel region using three different methods: modifying the environment variable `OMP_NUM_THREADS`, calling the function `omp_set_num_threads(...)`, and adding the `num_threads(...)` clause to the `omp parallel` construct. What is the order of precedence among these three methods?

Part 2

Sum of an array using OpenMP

Ex. 2

a) Write a C/C++ program (use the given skeleton code `sum-array-openmp.cpp`) that initializes an array `A` of N floating point numbers so that `A[i] = i` for all $0 \leq i < N$. Use a value of N sufficiently large (>1M) to see gains in parallel execution.

b) Add a second for loop that computes the sum of all elements in `A`.

c) Then, add a parallel region around the first loop, and parallelize it using `#pragma omp for`.

d) Now, parallelize the second loop with a `#pragma omp sections` construct having 4 sections. Each section should iterate over $N/4$ elements of `A`, find the sum of thse $N/4$ elements, and finally add it to the global sum of all N elements. Make sure to eliminate the race conditions **efficiently** using a single `#pragma omp atomic` operation in each section.

e) Query the number of hardware threads supported by your processor using the `lscpu` command, then execute your program with a timer using $1, 2, \ldots, P$ threads if your CPU has P threads. Compute the speedup/acceleration and efficiency for each execution.

Part 3

Parallel mergesort using OpenMP sections

The goal of this exercise is to sort an array of numbers using the mergesort algorithm in parallel using OpenMP sections. A skeleton code is already provided in the file `mergesort.cpp`. This code allocates and initializes an array `A` of `N` numbers as well as another temporary buffer array `temp` of the same size.

Ex. 3

a) Create a parallel region with 4 sections (or 8, if there is at least 8 hardware threads in your machine). Each section should sort `N / 4` consecutive elements of the array `A`. You can use either `std::sort` of the STL library or the `mergesort` function in the skeleton code.

b) In the same parallel region, after having finished these 4 sections for sorting, create 2 sections where each section merges two sorted arrays of `N/4` elements, to generate a sorted array of `N/2` elements. To do this, use the provided `merge` function that performs this merge operation in-place on `A`.

c) Finally, outside the parallel region, merge these two subarrays of size `N/2` to obtain the final sorted array `A` of size N.

d) Compare the sequential execution time with the parallel execution time using 4 (or 8) threads. What is the speedup/acceleration? What is the parallel efficiency?

Part 4

Computing the π

The π number can be defined as the integral of $f(x) = \frac{4}{1+x^2}$ from 0 to 1. An easy way to approximate this integral is to uniformly discretize the domain using N points with $s = \frac{1}{N}$ distance between two consecutive points:

$$\pi \approx \int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x \approx \sum_{i=0}^{N-1} s \times \frac{f(i \times s) + f((i+1) \times s)}{2}$$

We will write a C++ program that computes this approximation for π, then parallelize it using two different methods using OpenMP. A skeleton code is provided in the file `calcul-pi.cpp`.

Ex. 4

a) First, write a sequential code in the skeleton code that correctly computes the value of π using this formula.

b) We can then distribute this computation among P threads available in your machine. First, simply parallelize the main loop of computation using `pragma omp for`. Check the result. Is it correct? Why not (and leave it as is for now). Test the performance and acceleration using different number of threads $(1, 2, \ldots, P)$.

c) The second parallelization strategy is "by hand"; you are not allowed to use the OpenMP constructs `omp for`/`omp sections`. Each thread should execute a standard for loop, but with a different domain of iteration (begin/end) of size `N / P` depending on its thread id, and compute a partial value of π for its domain. Next, each thread will merge these partial values into the final value of π. In doing so, make sure that you avoid race conditions, using `omp atomic` clause. Your code should perform correctly and efficiently for any number of threads P. Test the performance and acceleration using different number of threads $(1, 2, \ldots, P)$.

d) Now, at the end of your `pragma omp for` directive, add the `reduction(+:pi)` clause, assuming that `pi` is the name of the shared valuable that accumulates partial results. Is the result correct now? Indeed, `reduction` clause does exactly the manipulation you did in the previous case behind the curtains (creating local accumulators, summing them up in each thread, then merging them with a single atomic operation), but automatically!.