TP - Introduction to OpenMP

Oguz Kaya, Atte Torri, Matthieu Robeyns
{oguz.kaya,atte.torri,matthieu.robeyns}@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 programme.cpp -o programme
```

Part 1

Hello World

Ex. 1

a) Write a program having a parallel region in which each thread prints its id.

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

Ex. 2

a) Write a C/C++ program 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) Now add a parallel region enclosing both loops, and parallelize each loop using a separate `#pragma omp for`.

d) Can we add a `nowait` clause to the first loop's `#pragma omp for`? Explain.

e) 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 `#pragma omp critical` and `#pragma omp atomic`.

f) 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, and plot these results.

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`. Be careful about the race condition here; you need to add an OpenMP clause to avoid it. 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` or the clause `reduction`. 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 either `omp critical` or `omp atomic` clause. Your code should perform correctly and efficiently for any number of threads P.

Part 5

Goldbach's conjecture

The goal of this exercise is to study different loop scheduling policies provided in the OpenMP library, in order to improve the performance of the parallel execution. A skeleton code is provided in the file `goldbach.cpp` for this exercise.

In number theory, Goldbach's conjecture claims that each even number greater than two is the sum of two prime numbers. The goal of this exercise is to test this conjecture with using parallel computation.

Dans cet exercice, vous êtes fournis un code séquentiel qui teste cette conjecture. Le code permet de trouver le nombre de paires de Goldbach pour un nombre pair donné i (c'est à dire le nombre de paires de nombres premiers P_1, P_2 tels que $P_1 + P_2 = i$) pour $i = 4, \ldots, 8192$. Le coût de calcul est proportionnel à i^2 dans l'itération i, donc pour obtenir une performance optimale avec plusieurs threads la charge de travail doit être distribuée en utilisant intelligemment la clause `schedule` de l'OpenMP.

The skeleton code enables to find the number of Goldbach prime numbers for a given even number. That is to say, the number of prime numbers P_1 and P_2 such that $P_1 + P_2 = i$ for $i = 4, \ldots, 8192$, $i \mod 2 = 0$. The computational cost is proportional to i^2 in the i^{th} iteration; therefore, to obtain an optimal performance in parallel execution, the loop's iteration domain should be distributed among threads intelligently using the `schedule` clause of OpenMP.

We ask you to perform the following tasks:

Ex. 5

a) Parallelize the sequential code using `omp for` without specifying a scheduling. What is the default scheduling strategy? What is the default chunk size for this scheduling strategy?

b) Parallelize the code using `static` scheduling strategy with a chunk size of 256.

c) Parallelize the code using `dynamic` scheduling without specifying the chunk size. What is the default chunk size for this strategy?

d) Parallelize the code using `dynamic` scheduling with a chunk size of 256.

e) Parallelize the code using `guided` scheduling without specifying the chunk size. What is the default chunk size for this strategy?

f) Parallelize the code using `guided` scheduling with a chunk size of 256.

g) Which scheduling and chunk size selection(s) gave the best results? Does this make sense with respect to the complexity of this algorithm?

Part 6

Parallel dense matrix-vector multiplication using OpenMP

The goal of this exercise is to write a programm that computes the multiplication of a matrix A of size $N \times N$ with a vector x of size N to obtain a vector b of size N:

$$b(i) = \sum_{j=1}^{N} A(i,j)x(j)$$

A skeleton code is provided in the file `matvec.cpp`, which allocates the vectors x and b as well as the matrix A. The matrix A is stored row-wise on a vector of size N^2, meaning that the element $A(i,j)$ is located at `A[i * N + j]` in the flat array `A`.

Ex. 6

a) Implement a sequential version in the provided section of the skeleton code, then measure the execution time.

b) Implement a parallel version using `omp for`, then measure the execution time.

c) Compute and print the parallel speedup/acceleration and efficiency.

d) Test the performance for all dimension sizes $N = 1, 2, 4, \ldots, 4096$. At what dimension size does the parallel version beat the sequential one? Modify the code so that for small dimension sizes where sequential execution is faster (meaning thread creation overhead becomes too costly), the code executes sequentially. You can do this either by making an explicit `if/else` branch, or using the `if` clause in OpenMP when creating your parallel region.