TP - Advanced 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

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.

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 $i = 4, \dots, 8192$, $i \mod 2 = 0$. 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. 1

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 2

Computing Fibonacci with OpenMP Tasks

In this exercise, we will compute Fibonacci[N] in parallel using OpenMP Tasks.

Ex. 2

a) Implement a recursive function that computes Fibonacci(N) using Fibonacci(N-1) and Fibonacci(N-2). Parallelize it using `#pragma omp task` and `#pragma omp taskwait` (do not forget to create a parallel region to have multiple threads). Test the correctness of your code. Test the speedup using different number of threads. Does it go faster?

b) Implement an iterative version that uses an array `Fibonacci` of size `N` to perform the computation. Set `Fibonacci[0]=0` and `Fibonacci[1]=1`, then in a for loop compute `Fibonacci[i]` using `Fibonacci[i-1]` and `Fibonacci[i-2]`. Next, parallelize the computation using OpenMP tasks by creating a separate task for computing each `Fibonacci[i]`. You need to add correct input and output dependencies to each task using the clauses `depend(in:...)` and `depend(out:...)`. Test the speedup of your program using different number threads. Does it go faster this time? Why/why not?

Part 3

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. 3

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.

Part 4

Parallel mergesort using OpenMP Tasks

Ex. 4

a) Parallelize the mergesort algorithm given in the skeleton code `mergesort.cpp` using OpenMP tasks. You should not need to rewrite or restructure the given sequential algorithm; it should be sufficient to create many threads (i.e., parallel region), then create tasks for recursive calls, finally wait for those tasks to finish before merging them. Limit the minimum task size (in terms of number of elements in the array) to a constant K (e.g., 100000) so that the recursion does not generate more tasks below this limit. Experiment with this to find the value K that gives the best performance.

Part 5

Parallel 2D prefix sum with OpenMP Tasks

In this exercise, we will compute the prefix sum of a 2D array $A[N][N]$ so that $B[x][y] = \sum_{(i,j)=(0,0)}^{(x,y)} A[i][j]$. An efficient way of doing this is through dynamic programming; for each $B[x][y]$, we can make sure to have computed $B[x][y-1]$, $B[x-1][y]$, and $B[x-1][y-1]$ beforehand, then compute

$$B[x][y] = B[x][y-1] + B[x-1][y] - B[x-1][y-1] + A[x][y]$$

. for all x, y. Indeed, this equation implies some depencies that need to be respected for an accurate computation (e.g., $B[x][y-1], B[x-1][y], B[x-1][y-1]$ must be computed before $B[x][y]$).

Ex. 5

a) Implement a sequential version of this code that first initializes the array $A[x][y] = x + y$, then computes B using two nested loops over x and y. Can you parallelize it using `omp for`? If yes, how much parallelization do you have (maximum number of active threads you can utilize)? If no, why?

b) Next, parallelize it using OpenMP Tasks this time, by creating N^2 tasks (one task for computing each $B[x][y]$), and adding necessary input/output dependencies. Measure the parallel execution time. Is it fast? Why/why not?

c) Finally, make a blocked version of your code with a block size K so that each task computes a tile of $K \times K$ contiguous elements of B (e.g., $B[0 \dots K-1][0 \dots K-1]$). You should create $(N/K) \times (N/K)$ tasks this time. Do not forget to add necessary input/output dependencies. To specify these dependencies, you can create and use a boolean array of size $(N/K) \times (N/K)$. You should have 4 nested loops in this version (two loops of size N/K iterative over block indices, two loops of size K iterating over a tile $K \times K$ (which constitutes one task).