TP - OpenMP Tasks

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 program.cpp -o program
```

Part 1

Computing Fibonacci with OpenMP Tasks

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

Ex. 1

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 2

Parallel mergesort using OpenMP Tasks

Ex. 2

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 3

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. 3

a) Implement a sequential version of this code with 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., $B[0 \ldots K-1][0 \ldots K-1]$. 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 block (which constitutes one task).

Part 4

Parallel matrix multiplication using OpenMP Tasks

The goal of this exercise is to compute the multiplication of two $N \times N$ matrices A and B in parallel using OpenMP tasks:

$$C = AB$$

We will use recursion to generate task parallelism. For this, we will consider the blocking

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}, A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

where each block is a $N/2 \times N/2$ matrix. With this blocking, the standard multiplication can does the following matrix multiplications and additions to compute each block of C:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Here, there are eight independent matrix multiplications, followed by four matrix additions. Strassen's algorithm reduces the number of multiplications to seven with the following formula:

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_2 = (A_{21} + A_{22})B_{11}$$
$$M_3 = A_{11}(B_{12} - B_{22})$$
$$M_4 = A_{22}(B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{12})B_{22}$$
$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12}$$
$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

We can then express the resulting matrix C as:

$$C_{11} = M_1 + M_4 - M_5 + M_7$$
$$C_{12} = M_3 + M_5$$
$$C_{21} = M_2 + M_4$$
$$C_{22} = M_1 - M_2 + M_3 + M_6$$

Ex. 4

a) Implement a matrix multiplication function that takes three matrices A, B, C of size $N \times N$, then divides each matrix into four submatrices of size $N/2 \times N/2$. For matrices, use 1D array/`std::vector` of size N^2, and use row-major storage to represent a 2D matrix: $A(i, j) =$ `A[i * N + j]`.

b) Implement a sequential function that multiplies two square matrices.

c) Implement a sequential function that adds two square matrices.

d) Implement a function that divides an $N \times N$ matrix into four submatrices of size $N/2 \times N/2$ each.

e) Implement a parallel version of the standard matrix multiplication using OpenMP Tasks. Make sure not to generate tasks for matrices below a certain threshold (just perform the multiplication directly).

f) Implement a parallel version of the Strassen's matrix multiplication using OpenMP Tasks. Make sure not to generate tasks for matrices below a certain threshold (just perform the multiplication directly).

g) Compare the parallel performance of these two implementations.