Task Parallelism with OpenMP

Universitat Autònoma de Barcelona

Matrix Multiply

This is an example of the usage of tasks for solving a **divide & conquer** algorithm for matrix multiplication. A divide and conquer design pattern transforms a single problem into sub-problems that can be solved independently. The cost of splitting the problem and merging the partial solutions should be cheap compared to the cost of solving the sub-problems. Eventually the sub-problems become small enough that serial execution is more efficient than dividing the sub-problem

In this case, the multiplication of two matrices of size **n×n** is split into eight sub-problems, each consisting on a multiplication of matrices of **n**/2×**n**/2. The description of the divide&conquer strategy is shown in the figure on the right.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$= \begin{bmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{bmatrix} + \begin{bmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{bmatrix}$$

8 multiplications of $n/2 \times n/2$ matrices.
1 addition of $n \times n$ matrices.

We provide a simple code that assumes that **n** is an exact power of two. Since the 2D matrices are represented using unidimensional vectors, the size of the original matrices (**N**) must be passed along the recursive calls to calculate the position of the rows in the matrices. The value **DQSZ** determines when a sub-problem is small enough to be solved serially. A C++ template is used to provide flexibility with the data type of the elements of the matrix.

The eight tasks could be executed simultaneously at the expense of using extra memory space for the partial results and adding the partial results in pairs in a final merge stage. The presented code reduces parallelism for the sake of saving memory space and saving a final matrix addition (instead, the addition is done on the fly using the memory space for the output matrix c). Therefore, four initial parallel tasks are generated with a **task** construct and then a **taskwait** construct is used to wait for their termination until the last four parallel tasks are generated.

The recursive **Divide & Conquer** strategy provides good data locality at multiple scales, and makes the algorithm ***cache-oblivious***, which means that the algorithm works reasonably well regardless of the actual cache structure.

Evaluar Ejecución Single-Thread

1. Evaluar el rendimiento *single-thread* de las versiones **clásica** y **divide&conquer** con tamaños **n**= 1024 y 2048. Para el segundo caso, probad con los valores DQSZ= 16 y 256.

2. Optimizar las dos versiones del programa **intercambiando** los **dos bucles internos** que realizan la multiplicación de matrices. Evaluad la mejora del rendimiento con los tamaños **n**= 1024, 2048 y 4096. Explicad las razones de esta mejora e indicad cuál es el cuello de botella del rendimiento en cada caso.

3. Explorar la estrategia de "***Register blocking***". Consiste en desenrollar simultáneamente los dos bucles externos para poder aprovechar la localidad temporal del algoritmo, reutilizar datos en registros y así reducir el número total de accesos a las matrices (y por tanto reducir el número total de accesos a memoria).

Paralelización MIMD

1. Paralelizar la versión **clásica** del programa usando las directivas **parallel** y **for**. Medid la mejora de rendimiento para **n**= 2048 y 4096. Verificar que el resultado del programa paralelo coincide con el del programa secuencial (asumiendo que se pueden producir pequeñas diferencias debidas a errores de redondeo).

2. Paralelizar la versión **divide&conquer** del programa usando las directivas **task** y **taskwait**. Medid la mejora de rendimiento para **n**= 2048 y 4096, y para DQSZ= 16 y 256. Verificar que el resultado del programa paralelo coincide con el del programa secuencial (asumiendo que se pueden producir pequeñas diferencias debidas a errores de redondeo).

3. Explicar las diferencias de rendimiento entre las dos versiones, y a partir de ellas encontrar el cuello de botella del rendimiento en cada caso. Predecid el tiempo de ejecución de cada versión para **n**= 8192 y luego verificar la predicción y sacar conclusiones.

Nota: en todos los casos se debe usar el compilador g++ versión 8.2, con la opción de optimización **–Ofast**. Recordad el uso de la opción **–fopenmp** para las versiones que usen OpenMP.