Concurrent Matrix Multiplication

Design

The purpose of the program is to provide an efficient approach to multiply matrices using the least time possible. In order to do so, I aim to provide a comparison between the convention approach to matrix multiplication i.e. serial approach, static approach of the OpenMP scheduler and the dynamic approach of the OpenMP scheduler. My code consists of 4 independent functions, which are designed to do independent tasks. Initially before all the function calls, the user is expected to input the size of the matrix he/she expects to be multiplied, which will be considered to be 'n'.

1. matrix_generation function:
   The function is designed to randomly generate integers and put them in independent matrices to be multiplied. The number of elements being generated depends on the size of the matrices entered by the user.
2. serial_multiply_matrices function:
   This function demonstrates the conventional matrix multiplication function where the elements of each row of the first matrix is multiplied by the elements of each column of the second matrix. The time taken to do this is calculated by taking a time reading before and after the operation is completed.
3. parallel_multiply_static function:
   This function demonstrates the matrix multiplication operation using OpenMP Static Scheduler. This function also uses the conventional logic for matrix multiplication, but uses the concepts of OpenMP to make the code execution more efficient and faster.
4. parallel_multiply_dynamic function:
   This function too demonstrates the matrix multiplication operation, but uses the Dynamic Scheduler in OpenMP to dynamically schedule the chunks to threads.

In both the OpenMP functions we are using the schedule clause of OpenMP For loops. The schedule clause defines how the corresponding loop iterations are split into contiguous non-subsets, named chunks, and how such chunks are allocated among the threads. In the context of their implied mission, each thread executes its allocated chunk(s).

Compilation: g++ -fopenmp MatrixMul.cpp -o MatrixMul

Implementation and Efficiency:

The serial implementation of the Matrix Multiplication logic is straight forward and gets the job done. But the code efficiency begins to fade with increasing size of matrices and hence we had to implement approaches in OpenMP that parallelize the process using multithreading. An OpenMP program usually starts with a single thread called the Master thread and then forks child processes in order to split its work off into smaller chunks, which will be delegated to the forked child threads. There is a collection of variables available to each thread, which the thread might use or update during its execution. At the end of the parallel region, the child threads terminate and give back the control to the Master thread, which continues execution and finishes the task assigned.

My code is developed on the basis of concepts of Scheduling and Collapse function available in OpenMP.

Let us consider the **parallel_multiply_static** first. The function makes use of the general construct for OpenMP parallel for Loop, followed by the scheduling options along with chunk size, the collapse keyword, private variable and shared variables. When **schedule(static, chunk_size)** is specified, loop iterations are divided into chunks of size **chunk_size**, and the chunks are assigned to the threads in a round-robin fashion in the order of the thread number. In this case, the chunks are divided to threads in a serial fashion and if a thread finishes its operation, it has to wait until its turn arrives again, after assignment is completed for all the other threads in the queue. **Static scheduling** is a work-sharing system for deception which fits best with unbiased loops that have a fairly constant expense per iteration.

As the approach uses the divide and conquer approach of things, the approach is efficient and makes good use of available resources. In addition to this, the approach uses the **collapse** keyword to let the compiler know the number of loops associated with the loop construct. Using this clause, it enables the code to be collapsed into one large iteration space, which is then divided accordingly by the schedule clause. In our case here, we have used the collapse clause on the loop i and j, hence the iteration space is collapsed into a single loop and executed. The logic also uses variables i, j, k as private in order to prevent parallel access and write by threads. Whereas, the matrices ‘a’ and ‘b’ are declared as shared resources across the threads.

Let us now consider the **parallel_multiply_dynamic** function, where we use the dynamic approach of the OpenMP scheduler for parallel ‘for’ loops. The dynamic construct also uses a similar approach as the static schedular, but the **approach of threads picking up chunks varies**. The chunks are divided according to the chunk size mentioned and then the chunks are assigned to the spawned threads dynamically i.e. distributed to threads as the threads request them. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed.

In this setting, the number of iterations being supplied to each thread **might vary on the workload / number of iterations** present at the time of assigning. Hence, threads might undergo different workload. By default, the system will decide how many threads to fork during runtime. This approach is more efficient as none of the resources as wasted at any point and resources are allocated to threads immediately when they finish their assigned task.

The approach coupled with the **collapse** clause, enables the compiler to efficiently combine the logic of both the for loops to construct an equivalent loop, which executes faster. Once the for loops are collapse into a large iteration space, it is then divided accordingly into chunks and assigned to threads using dynamic scheduling. As seen in the previous case, this approach too uses i, j, k as private variables to each thread and a, b as shared variables which is accessible to each thread.

Hence, I could conclude that the **chunk size and the scheduling approach decide the efficiency** of the algorithm in each case. It was observed that during most of the times for various scenarios, the dynamic approach was the most efficient and fast. The static scheduling might be slow in a few cases as some loops are unbalanced, and hence some iterations take much longer than others. Also, I could observe that having a **lower chunk size gives a reasonably faster algorithm.** This might not be significant for smaller matrices, but become significant if and when the size of the matrices increases.

References:

1. OpenMP Application Program Interface Version 4.5 November 2015 Copyright © 1997-2015 OpenMP Architecture Review Board

2. C.K.Balaji Department of Computer Science, Government Arts & Science College, Sivakasi, S. Kartheeswaran MCA Department, Kalasalingam Academy of Research and Education, Krishnankoil - Analyzing the Matrix Multiplication Performance in Shared memory processor Under Multicore Architecture Using OpenMP, International Journal of Pure and Applied Mathematics, Volume 119 No. 15 2018, 3249-3256.