1. **Problem Statement**

Given the matmul function in *matmul.c*, use OpenMP directives in the function to improve the performance. This will demonstrate how parallel processing can be leveraged to improve processing time for other large computational problems that take significant processing time. The scalability of OpenMP will also be examined for this use case.

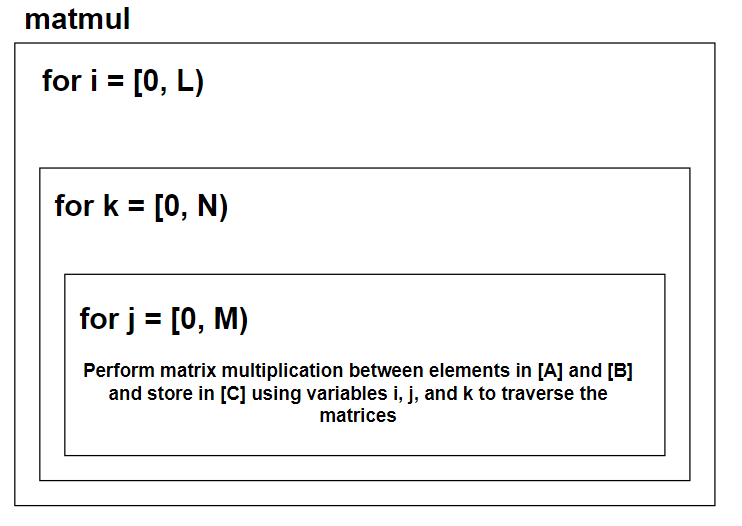
1. **Approach**

Because the given function was coded in C, the decision was made to use NetBeans IDE to implement changes while using the Windows Shell to quickly compile the code and execute the program. Because OpenMP requires compiling with the –fopenmp option to allow the compiler to process OpenMP directives, the matmul function could be quickly tested with and without the new parallel constructs simply by compiling and running with and without the flag. By varying the problem size and testing, the performance with and without OpenMP can be observed and trends plotted. The test analyses:

* 1. Performance change when varying size outer task pool (Size of L)
  2. Performance change when varying problem size of inner loop (Size of M)
  3. When problem size is too small and OpenMP overhead is too large
  4. Overall trend of runtime improvement with growing problem size

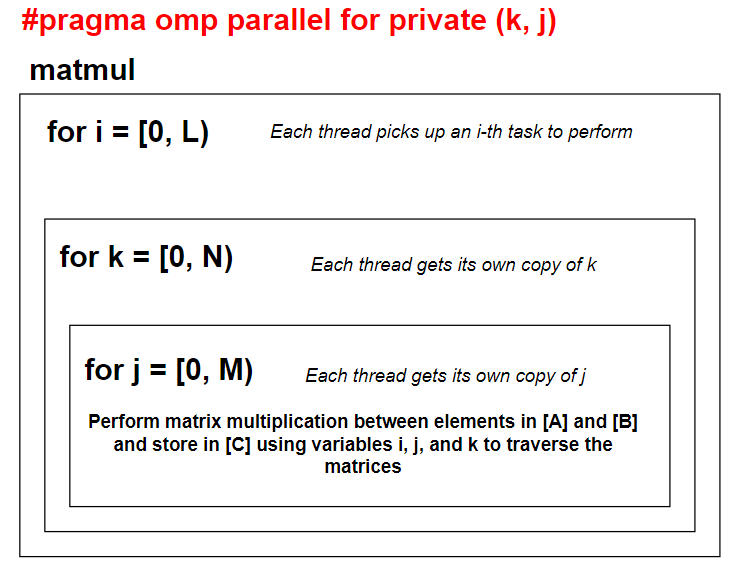
1. **Solution**

With the goal of improving the processing time for the matmul function, the initial task for this problem domain was to get an understanding of how the matmul function works and demonstrate parallel processing through OpenMP. Because the global constants, L, M, and N, are used to define the size of the matrices within the problem, these constants also control the problem size and were altered in each test run to determine changes in performance. Within the matmul function, there is a primary for-loop with two nested for-loops. This can be illustrated as shown in the following illustration (Figure 1).



**Figure 1. Illustration of matmul function structure**

Utilizing this algorithm, OpenMP can leverage extra threads to improve processing time by performing independent tasks in parallel. This can be performed by adding an OpenMP pragma for a parallel-for construct to the outermost for-loop. When compiled with the –fopenmp option, the resulting program will create a team of threads to process each of the iterations from 0 to L for the outer loop. However, because OpenMP utilizes shared memory, this creates a problem. If the inner variables k and j are shared, this can create a slew of unaccountable behavior. As such, the variables k and j must be kept private in each thread, allowing each thread to loop through and manage its own copy (Figure 2).

****

**Figure 2. Illustration of matmul function with OpenMP parallel construct added**

Now that the problem domain was understood and the proper OpenMP directive has been added, all that remained was to compile and test the program to determine performance. By performing runs compiling with and without the OpenMP compiler flag, the following two run results were generated (Figure 3).

**Figure 3. Performance tests using OpenMP on matmul function**

The first run was performed using the following parameters:

* L = independent
* M = 400
* N = 400

The second run was performed using the following parameters:

* L = independent
* M = 400
* N = 800

In both cases, there was a significant improvement in performance runtime. In Run 1, there was an average of 3.5x faster performance with OpenMP. In Run 2, there was an average of 4.3x faster performance. In both cases, however, there was a minimum test case where the number of tasks was small enough that the difference between OpenMP overhead to processing efficiency yielded very minimal improvement. This led to examining how the internal problem size affected performance (Figure 4).

**Figure 4. Examining the runtime performance as a function of the internal problem size**

As shown in Figure 4, as the internal problem size grew, the more significant the improvement was observed. Plotting these results on a logarithmic scale, the overall performance improvement trend can be further evaluated (Figure 5).

**Figure 5. Overall evaluation of OpenMP on matmul function as function of total internal problem size (M \* N)**

The results as illustrate in Figure 5 was very valuable to these tests. By quick examination, it can be determined that the improvement of utilizing OpenMP parallel processing over the original code which does not utilize parallel processes happens on a logarithmic scale with the implementation becoming more and more valuable as the internal problem size grows. Also equally important is noticing that at very small problem sizes, using OpenMP yields slower performance. Again, this has to do with the overhead of fork/join actions needed when utilizing additional threads. With this information, the decision was made to make one final improvement to the matmul function which adds the condition that the internal problem size of N\*M must be greater than 2000 for OpenMP directives to be executed. This dictates that at problem sizes smaller than 2000, the function will execute sequentially, and for problem sizes larger, OpenMP will utilize multi-threading parallel processes to execute the function (Figure 6).

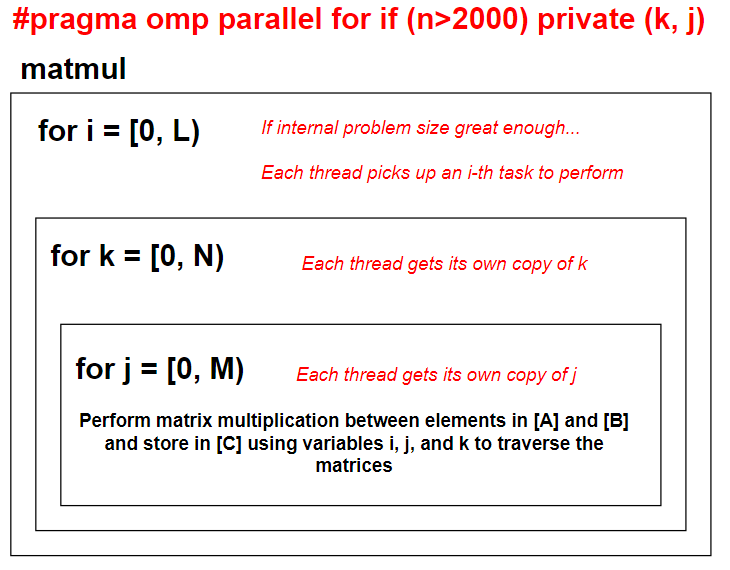


Figure 6. Illustration of final OpenMP implementation in matmul function

This concludes the improvements made to matmul.c. Additional comments were updated throughout the source code to document changes made. The internal problem size constant value of 2000 was stored as a global constant, OMP\_MATMUL\_TRIGGER, at the top of the source code for ease in future edits. A sample compiled program can be found in the attachment along with this report along with the edited source code.