Neighbor list implementation

In our implementation of the neighbor list, each molecule has an associated list of other molecules whose center of mass is within a spherical shell defined by a radius r_n and a skin thickness δ. This list is rebuilt from scratch periodically and is updated every time an MC move is accepted.

The reconstruction of the neighbor list is an expensive operation, as it involves calculation of pairwise distances. In order to speed up this operation, we used an algorithm that takes advantage of the *task* feature included in OpenMP.

Tasks are used to parallelize the execution of units of work generated through recursion. The specification of a task is conducted using the *task* directive, which encloses the code that will be considered as a task. When a thread encounters a task construct, it may choose to execute the task immediately or place it in a pool of tasks. Other threads in the team might execute tasks out of the pool until the pool is empty.

In the context of the neighbor list, tasks are defined by considering contiguous regions of the symmetric molecule index matrix, whose size is $N \times N$, as shown in Figure 5.2. Specifically, a task can be either the identification of one of these regions or the construction of the neighbor lists of the molecules associated to these regions. The shape of such regions can be triangular or rectangular. Triangular regions are distributed along the matrix diagonal, while rectangular regions occupy the rest of the matrix. To avoid race conditions, a recursive algorithm processes the triangular regions first followed by the rectangles. The optimal size of the triangles or rectangles was determined empirically, and it was determined that the base cannot be less than 32 molecules. The use of tasking improves the load balancing of the generation of the neighbor list, as all threads can take tasks from the pool at any time, minimizing the idle time.

Important subroutines:

SUBROUTINE Neighbor_Initialize(this_box)

Initializes neighbor lists for each molecule in this_box. This function is called at the very beginning of the simulation (i.e. in main.f90) or when the neighbor list needs to be reconstructed (i.e. for a GEMC simulation, it could be in the GEMC driver within an if statement that checks if the current MC step is a multiple of the frequency of the neighbor list reconstruction)

SUBROUTINE Triangle(A, B, is, this_box)

This subroutine aims to identify the triangles distributed along the diagonal of the NxN matrix. Initially, it takes the bounds upper triangular region of the matrix (black region in Figure 5.2) as inputs using A, B. Then, it splits this triangle in two at N/2. This division generates two triangles and one rectangle. The first triangle has a base and height of 1 to N/2. The second triangle has a base and height of N/2 to N. The rectangle has a base and height of N/2.

Recursively, it this function calls itself and takes as inputs the two triangles identified from the previous iteration. The same splitting process will continue until the identified triangles are small enough. Once this point is reached, the Neighbor_Build_Triangle function is called.

SUBROUTINE Rectangle(A,B, is, C, D, js, this_box)

This subroutine aims to identify the rectangles (or squares) within the black region of Figure 5.2. It works similarly to the triangle subroutine.

SUBROUTINE Neighbor_Build_Triangle (A, B, is, this_box)

This function measures the distance between two molecules, applies minimum image convention and decides whether a molecule must be added to the neighbor list of the other.

SUBROUTINE Neighbor_Append(im, is, jm, js, this_box)

This function appends a molecule jm that was found within the neighbor list region of the reference molecule im.

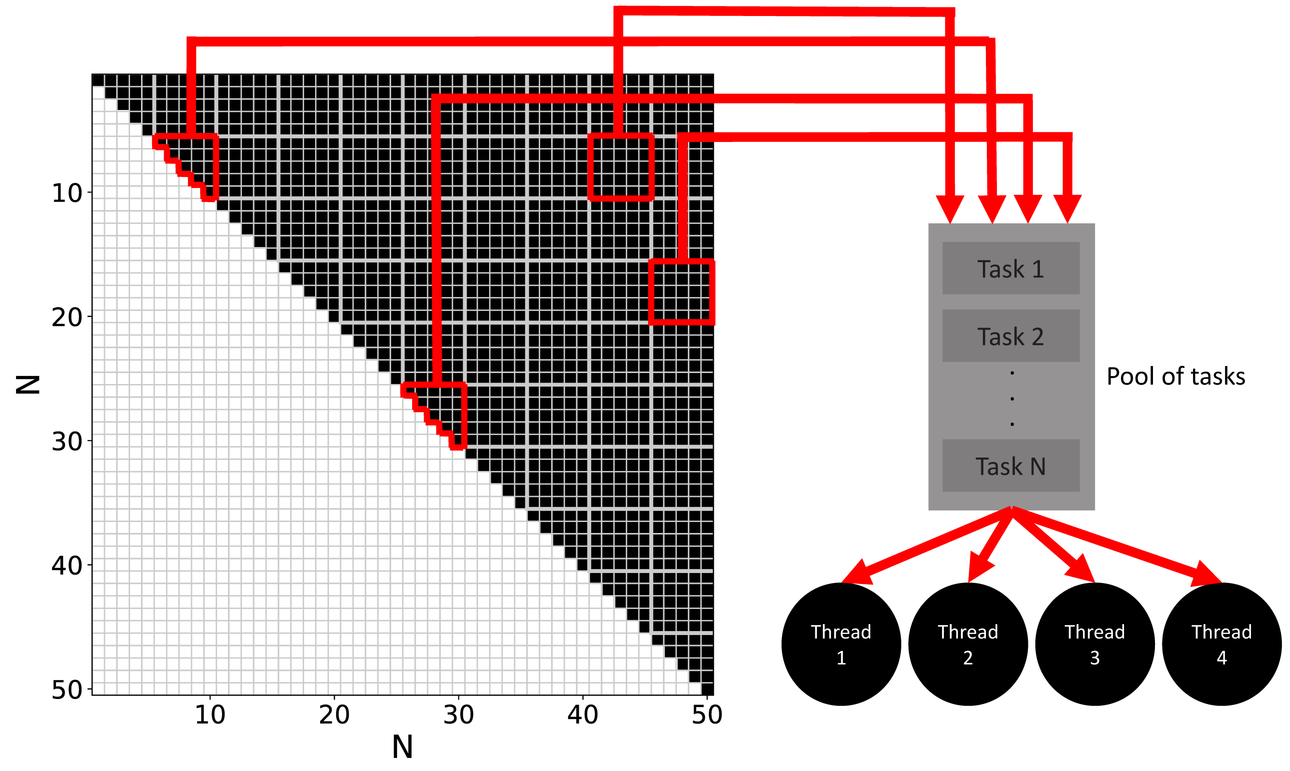

Figure 5.2: Diagram that illustrates the construction of the neighbor list required in the AVBMC method using the *tasking* method included in OpenMP.