Molecular Dynamics Task Graph

Amanreet Bajwa

July 25, 2011

1 Abstract

Molecular dynamics codes often deal with large amounts of data with many dependencies which result in greater execution time for the code. These dependencies increase the execution time of the code because the code is run serially rather than in parallel. In order to improve on the codes performance I have parallelized the code to a greater extent by doing a full sparse tiling [2] on three main molecular dynamics loops. Additionally, to better visualize these loop dependencies I have created an algorithm that graphs and labels each iteration of each node, shows the dependencies (edges) between nodes of different loops, and labels each node so that the tiling can be seen. The results of the tests run between the code that is not reordered using the partition graph coloring algorithm and the code that is have shown that when partition graph coloring is used the average parallelism increases. The results have also shown that when partition graph coloring is not used the code runs linearly with an average parallelism of 1.

2 Introduction

Moldyn is a molecular dynamics simulation [1] that models the interactions between molecules. The main moldyn code consists of three loops. The first one loops over the molecules, the second one over the interactions between those molecules and the third loops, again, over the molecules. These loops all use the same set of molecules so there are many dependencies between them. These dependencies between loops occur when there is an interaction between two different molecules. For example, we have the interactions array in Table 2 that is filled with pairs of molecules that interact. In column 0 we have the pair 0 and 1, this means that molecule 0 and molecule 1 interact with each other. This interaction creates a dependency from the first (molecules) loop to the second (interactions) loop from molecule 0 in the molecules loop to iteration 0 in the interactions loop. There is another dependency from molecule 1 in the molecules loop to iteration 0 in the interactions loop. These are mirrored between the interactions loop and the third (molecules) loop as well so there is a dependency from iteration 0 in the second loop to both molecule 0 and molecule 1 in the third loop. These dependencies generate edges between the three loops that can be seen in the dependency graph in Figure 3.

In order to help with the parallelization of these loops there is an algorithm that tiles these loops. The tiles and the dependencies between the loops are hard to visualize in one's mind and part of my project helps solve that problem. The dependency graphs generated by my algorithm create a graph that shows the dependencies between each of the three loops. The graph is also color coded according to the tile a node is put into. From the graph, one can see which nodes in which iterations are in certain tiles.

Another part of this project is the task graph interface. The task graph interface is used to create a graph that shows dependencies between tasks. In the case of moldyn, each task represents a tile and the task graph

Table 1: Information about the graphs used

input	*molecules*	*interactions*
test.graph	10	17

Table 2: Example interactions array

0	*1*	*2*	*3*	*4*	*5*	...	*16*
0	0	0	0	0	1	...	7
1	2	3	4	6	2	...	8

shows a dependency between the tasks/tiles if there is a node used in both tiles. This is another handy visualization tool that shows how dependent the three loops are on each other. The task graph interface allows the user to specify a parallelization engine to be used on the code. The engines include OpenMP, tbb, cilk, cnc, and pthreads. This returns different time results depending on the algorithm. Adding the moldyn algorithms into this interface was another part of the project that I worked on. To the best of our knowledge molecular dynamics codes have never been parallelized this way before.

The use of partition graph coloring on the moldyn code when compared to the code without resulted in what we expected. The average parallelism of the code when run without the partition graph coloring ended up being 1 for every test run on every file in Table 3. The results when partition graph coloring was used shows a higher average parallelism for each test, as expected. These results are displayed in Figure 7 and Figure 8. Figure 7 compares the average parallelism to the partition size while Figure 8 compares the average parallelism to the tile size.

3 Fine Grained Dependence Graph

In the dependency graph (see Figure 3) the nodes represent an iteration of a certain loop and are labeled as such. The edges represent a dependency between the iterations in different loops. There is an edge from the first loop of the molecules loop to the interactions loop if that molecule is involved in that particular interaction. These edges are reflected between the second and third loop. If there is a directed edge from the first loop to the second, for example, from node 1 in the first loop to interaction 5 in the second loop then there will be a directed edge from interaction 5 to node 1 in the third loop as well.

Figure 1 shows the algorithm that creates this fine grained dependence graph. The algorithm begins with three for loops that add each of the nodes from the three moldyn loops to the dot file that creates the visualization. Each iteration of a for loop is its own node in the graph. The loops also label the nodes with their loop number and iteration number. For example, if we are looking at loop 1 (the interactions loop) at iteration 2 the label on the node would be 1,2. These loops also set the color of the node according to which tile they have been placed in. After the three loops there is one final loop that iterates over the interactions array in order to create the directed edges between the three loops. This loop starts by getting the two molecules that interact with each other at the current iteration of the interactions array. At this point the directed edges from the first loop (0 - the molecules loop) to the second loop (the interactions) are formed. An edge is created for each of the two interaction molecules in the first loop to the interactions loop at the current iteration. After this the two directed edges from the current interaction node to the third loop (the molecules) are created. The interaction node has a directed edge to each of the two molecules in the third loop that are involved in the interaction. The pseudocode in Figure 1 shows the four loops involved in the algorithm. An example of a two tiled graph generated by this algorithm can be seen in Figure 3.

```
// creates labels for each iteration of the
// first loop in the moldyn algorithm
for each molecule
  add a line to the dot file that creates
  and labels the molecule in the first loop

// creates labels for each interaction
// represented in the second loop of the
// moldyn algorithm
for each interaction
  add a line to the dot file that creates
  and labels the interaction node

// creates labels for each iteration of the
// third loop in the moldyn algorithm
for each molecule
  add a line to the dot file that creates
  and labels the molecule in the third loop

// creates an edge from the first loop to the
// second loop (interactions) for each pair
// of molecules that interact and then creates an
// edge from the interactions loop to the
// same molecules in the third loop
for each interaction
  add the edges from the first loop for each
  interaction pair:
    (0, node1) -> (1, interaction)
    (0, node2) -> (1, interaction)

  add the directed edges from the interactions
  loop to the last molecules loop:
    (1, interaction) -> (2, node1)
    (1, interaction -> (2, node2)
```

Figure 1: Pseudocode for the tiled dependency graph dot file creation.

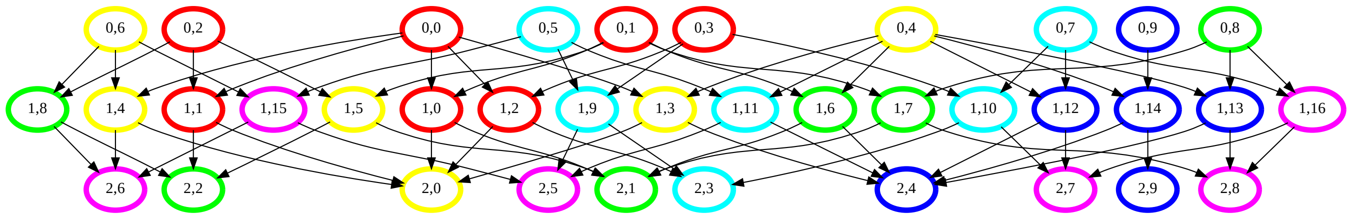

Figure 2: test.graph's dependency graph with tiling (6 tiles)

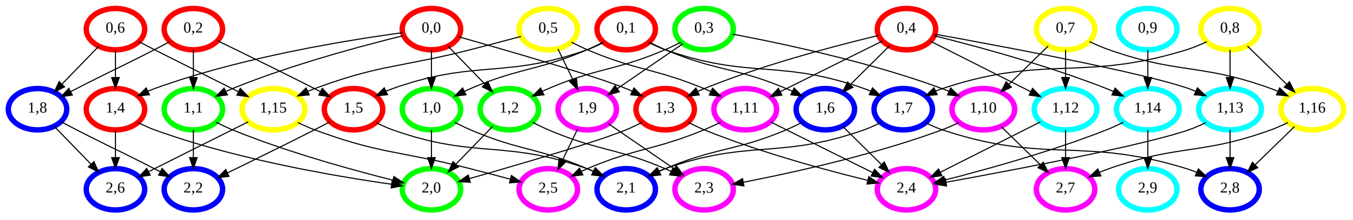

Figure 3: test.graph's dependency graph with tiling (6 tiles) after partition graph coloring

```
// initialize the different to the max
// difference there can be between the
// hues of the colors
colors array = {}
difference = 1/numberOfColors

// add to the colors array each new hue by
// adding the difference to the previous hue
for numberOfColors times
  current color = previous color + difference
```

Figure 4: Pseudocode for the distinct color generating algorithm.

4 Labeling of Tiles

The number of tiles created by the program is a parameter of taskMaster, that is the program that creates task graphs. Anywhere from 1 to the number of interactions can be entered for this parameter which poses a problem when creating the fine grained dependence graph. Each tile in the fine grained dependence graph needs to have a distinctly visible color so that the tiles are easily differentiable. If there are not enough colors available to color code each tile the tiles are hard to discern from one another, that poses a problem.

This problem is where the labeling algorithm comes in, each tile is labeled with a color. The algorithm is used to create an array of easily differentiable colors, enough to color each tile differently. The colors are easily differentiable when there is high contrast between then. The algorithm uses the HSV color values which can be used with dot. HSV specifies the hue, saturation, and value of a color. The saturation as well as the value are kept constant at 1.0000 when creating new colors for the tiles. Just adjusting the hue provides a wide range of distinct colors. Another option would have been to generate RGB values. The problem with using RGB values is that they have the ability to create a lot of shades of grey which makes the possibility of coming up with an indistinct color more likely. Adjusting the hue, on the other hand, does not create these shades of grey and therefore creates colors that are easy for the human eye to discern. The values for the hue can go anywhere from 0.0000 to 1.0000. The further apart the numbers, the more distinct the colors. The algorithm starts with an array that contains the colors with only the hue 0.0000 in it. It also has a variable that holds the difference to use when generating new values which is initialized to

$$\frac{1}{numberOfColors} \tag{1}$$

since this is the maximum possible difference between each hue. The rest of the algorithm is simple. The loop iterates as many times as needed for how many colors are required. With each iteration of the loop another value is added to the colors array. The new value is the sum of the previous value plus the difference.

$$colorArray[i] = colorArray[i-1] + difference \tag{2}$$

This value is stored and the loop goes on to the next iteration. This algorithm generates just enough colors for the current tiling. There is no limit on the number of different colors that can be generated by the algorithm but if a number too high is used then the colors start to blur together. At 20 different colors a few of the colors start to look too similar to be easily differentiable.

The result of this process can be seen in the pseudocode in Figure 4.

5 Coarse Grained Dependence Graph

The task graph for moldyn is used to show the dependency between each task. In moldyn each task is represented by a tile. The already implemented task graph interface was used to create a program to generate task graphs for moldyn simulations. While the program is doing the tiling an edge is created

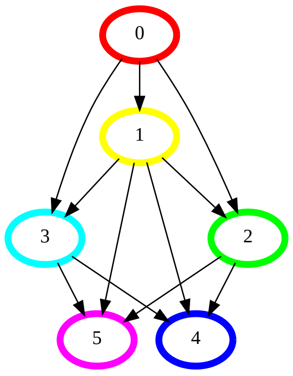

Figure 5: An example task graph without partition graph coloring

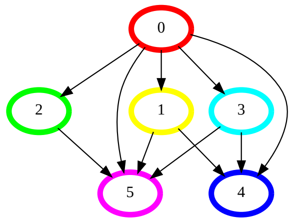

Figure 6: A task graph with the partition graph coloring algorithm applied

between tiles every time there is a dependency between the tiles. An example of this could be if a molecule in the first loop (the molecules loop) is in tile 1 but it is involved in an interaction in tile 2 but it makes more sense to keep that molecule in the first tile. This type of situation would create a dependency between the tiles. In the dot representation of the graph only one directed edge is generated between tiles/tasks regardless of how many dependencies are between them. For each of the tasks generated, the work performed by those tasks is to run the actual moldyn computation code and print out the results.

6 Partition Graph Coloring

In the main driver, taskMaster, a flag is provided for the option to create a partition graph, color it, and reorder the tiles in order to provide better parallelism [2]. This process helps to reorder the tiles in a way such that they can run in parallel more efficiently. In order to this a color graph is created first. The color graph contains a node for each seed partition. There is a loop over the interactions array in which each pair of molecules is checked to see if they are in different tiles. If they are in different tiles, an edge is added between the tiles the molecules are in. For example if there is an interaction between molecule 0 which is in tile 4 and molecule 1 which is in tile 2 there is an undirected edge from node 2 to node 4 in the color graph. After a color graph has been created it is colored using a graph coloring algorithm. This algorithm colors the graph with the minimum colors possible where no two connected nodes are the same color. This colored graph is then given to a function that reorders the nodes in the graph according to their color and returns a new tile ordering for the nodes. At this point the iterations of the moldyn loops can be modified to be in their new tiles. The resulting task graph shows more parallel computation is possible and the average parallelism is computed to prove this. An example of this can be seen in by the differences in Figure 5 and Figure 5. In Figure 5 there aren't many nodes that can be run in parallel as can be seen by the lack of horizontally adjacent nodes. In Figure 5 there are more adjacent nodes than in Figure 5. Notice that 2, 1, and 3 lie horizontal to each other and can all be run in parallel and 5 and 4 are the same way in Figure 5.

Table 3: Information about the pdb files used

input	*size*
1xib.pdb	306.7 KB
3Q8X.pdb	988.2 KB
3NZ8.pdb	1.1 MB
3RFZ.pdb	1.6 MB
2IA5.pdb	2.5 MB

In this case there is only one node (node 0) that cannot be run in parallel with any other. In the case of the graph before the coloring (Figure 5) there are two nodes, 0 and 1 that are not run in parallel with any others.

7 Calculating the number of tiles that fit into L1 cache

In order to get the best results from running tests on a certain machine the tests were optimized so that a tile could fit into L1 cache. L1 cache was used because the files used to run the tests were small enough that all the tiles could fit in L2 cache already. The first step was to figure out on average how many unique molecules were involved in a tile. I ran multiple tests on different graphs and came up with an equation

$$f(x) = 0.33966x + 7.5 \tag{3}$$

where x is the number of interactions and f(x) is the average number of unique molecules accessed in those interactions. At this point we know that 72 bytes is taken up by each interaction. In order to figure out the memory footprint of each tile the equation

$$mem = f(x) * 72 mem = (0.33966x + 7.5) * 72 \tag{4}$$

is used. The L1 cache for a machine called carrot, that all the tests were run on, is 32KB. In order to fit a tile into L1 cache on carrot it must be less than 32KB. From this data we can use the equation above to determine how many interactions can fit in the L2 cache and in turn how many tiles should be used when running the test. If we solve for x we get

$$x = ((mem/72) - 7.5)/0.33966 \tag{5}$$

Substituting in the amount of memory in L1 cache, 32KB for example, we get

$$x = (((32 * 1024)/72) - 7.5)/0.33966 = 1317.82 interactions \tag{6}$$

From this we can determine the number of tiles needed so that each tile can run on L1 cache by dividing the total number of interactions by the number of interactions calculated above.

$$tiles = totalinteractions/interactionsincache \tag{7}$$

This process allows for the tests to be optimized for a given computer. The process was used to calculate the number of tiles for each of the pdb files used for testing (shown in Table 3). Each of the files had its own set of tile sizes to optimize the tests.

8 Partition Graph Coloring Results

The results show that average parallelism was greatly affected by the partition graph coloring. The results of tests run on the five files in Table 3 show that the execution of the moldyn code is linear without the partition

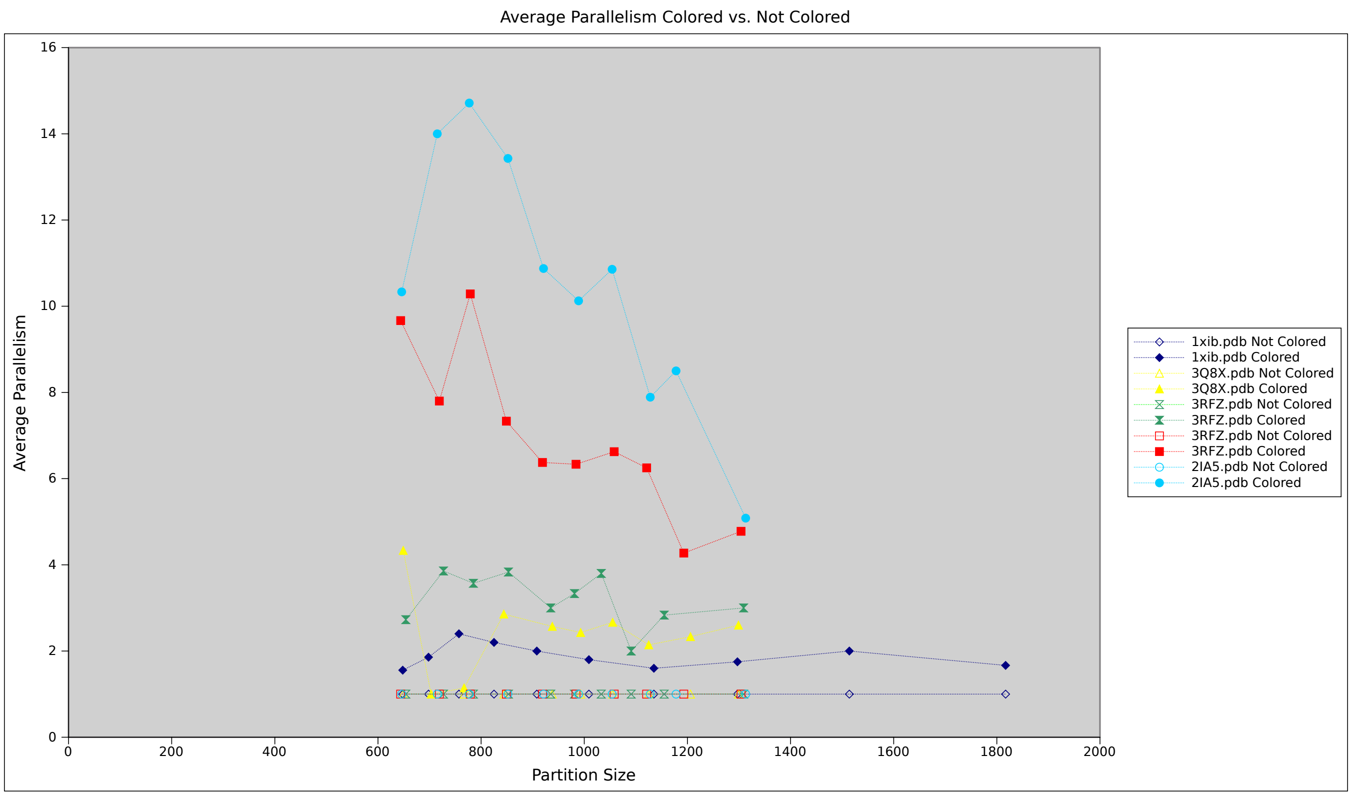

Figure 7: Results of the partition graph coloring with Partition Size

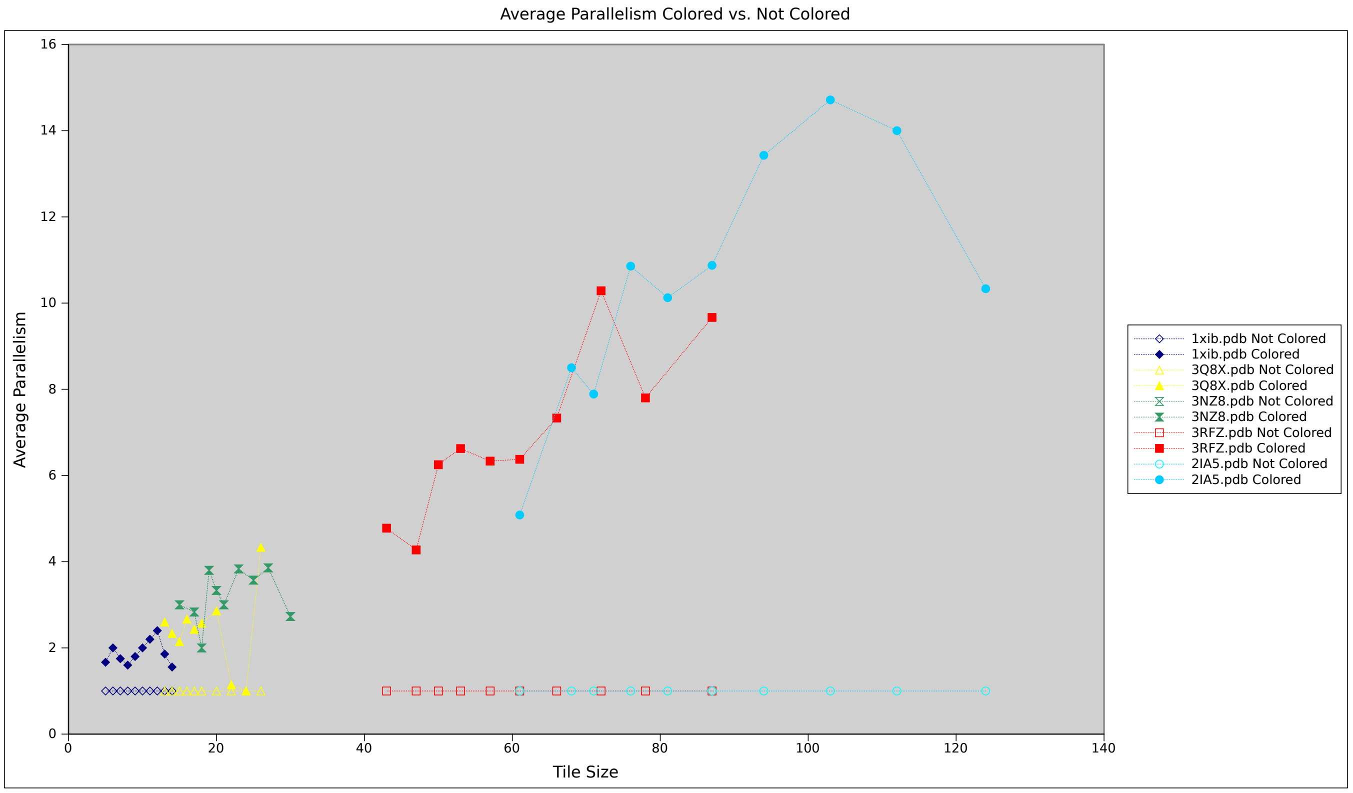

Figure 8: Results of the partition graph coloring with Number of Tiles

graph coloring. Each of the five files have been linear in execution for each of the 10 tests performed without the partition graph coloring. With the partition graph coloring algorithm applied the average parallelism has increased. For example, the results in Figure 7 display the results for the file 3RFZ.pdb with a dotted red line connecting red squares. The line with the not filled in squares represents the results for that file without the partition graph coloring, all of these results lie on a straight line with an average parallelism of 1, which means they run linearly. The results with the partition graph coloring are represented with the filled in red squares. As you can see the average parallelism (y-axis) is much higher with the partition graph coloring. These same results are also displayed in Figure 8 with the x-axis as the tile size rather than the partition size. The results are as expected since the average parallelism increases with the partition graph coloring algorithm applied. A higher average parallelism is good because that means that more parts of the code are running in parallel which makes the execution time faster.

9 Conclusion

I learned a lot of new things from this project. I learned about average parallelism and sparse tiling. I never understood how to measure the amount of parallelism of code before learning what average parallelism was and how to calculate it. I also learned what sparse tiling is and what the applications of it are. My results showed that using the partition graph coloring algorithm on the loops in the moldyn computation can increase the overall average parallelism of the code. They also showed that the average parallelism of the code without the partition graph coloring algorithm is 1 which means it runs serially. Future work for this project could include running tests to see if the overhead of partition graph coloring is less than the actual benefit of the algorithm to see if it is worth performing. Also, creating an OpenMP version of the moldyn code to compare the execution time of the computation with sparse tiling to see which performs better overall would be beneficial to finish.

References

[1] K. S. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[2] M. M. Strout, L. Carter, J. Ferrante, J. Freeman, and B. Kreaseck. Combining performance aspects of irregular gauss-seidel via sparse tiling. In *15th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, College Park, Maryland, July 25-27, 2002.