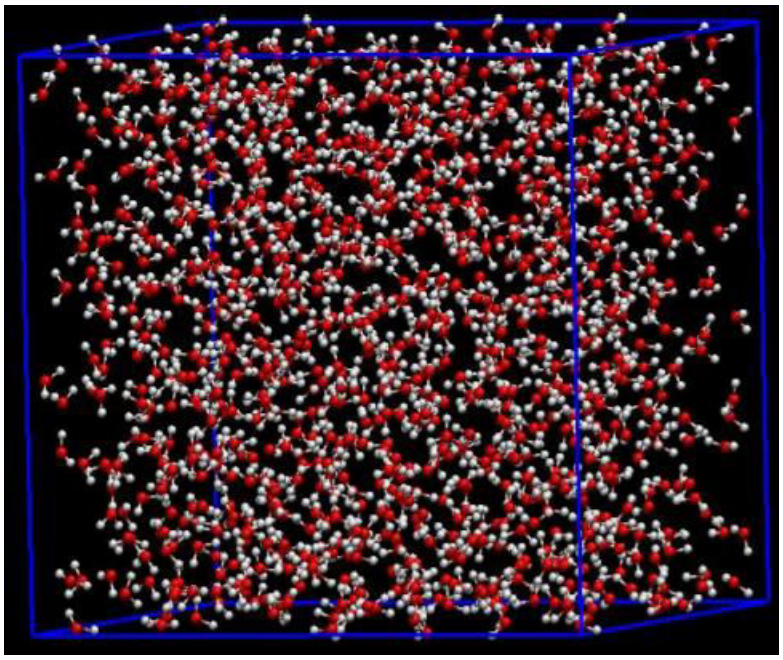
Calculation of Energy of configuration for a Box of Water Molecules

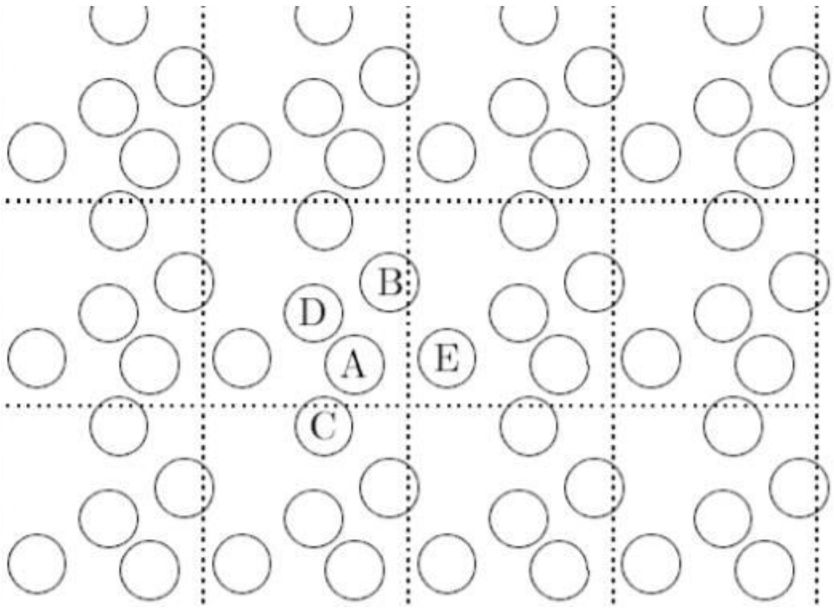
By - Hrithik Rai Saxena



Problem Statement - To parallelize the serial code using **MPI**(Message Passing Intereface) for calculating the energy of configuration for a box of water molecules and comparing its efficiency against different settings for number of processing threads and computing units.

Given - Position of each atom , Size of the Box, **.gro** input file(trajectory generated by a software called **Gromax**, used for molecular dynamic simulation) for three settings namely 1k.gro(1000 molecules), 10k.gro(10,000 molecules) and 100k.gro(100,000 molecules), Serial code for calculating the Energy Configuration.

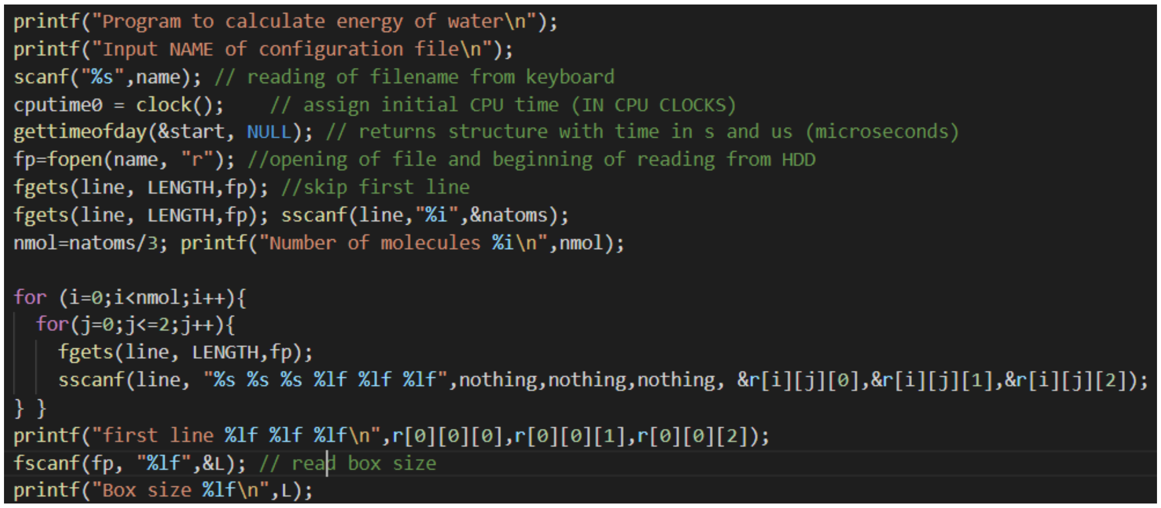
Note - Periodic Boundry Conditions are used wherein the boxes will be replicated, each surronded by its own replica such that the configuration is not disturbed by molecules surpassing the boundries of the box and ensuring box integrity using their replicas.



Periodic Boundry Conditions

<u>Program components to be parallelized:</u>

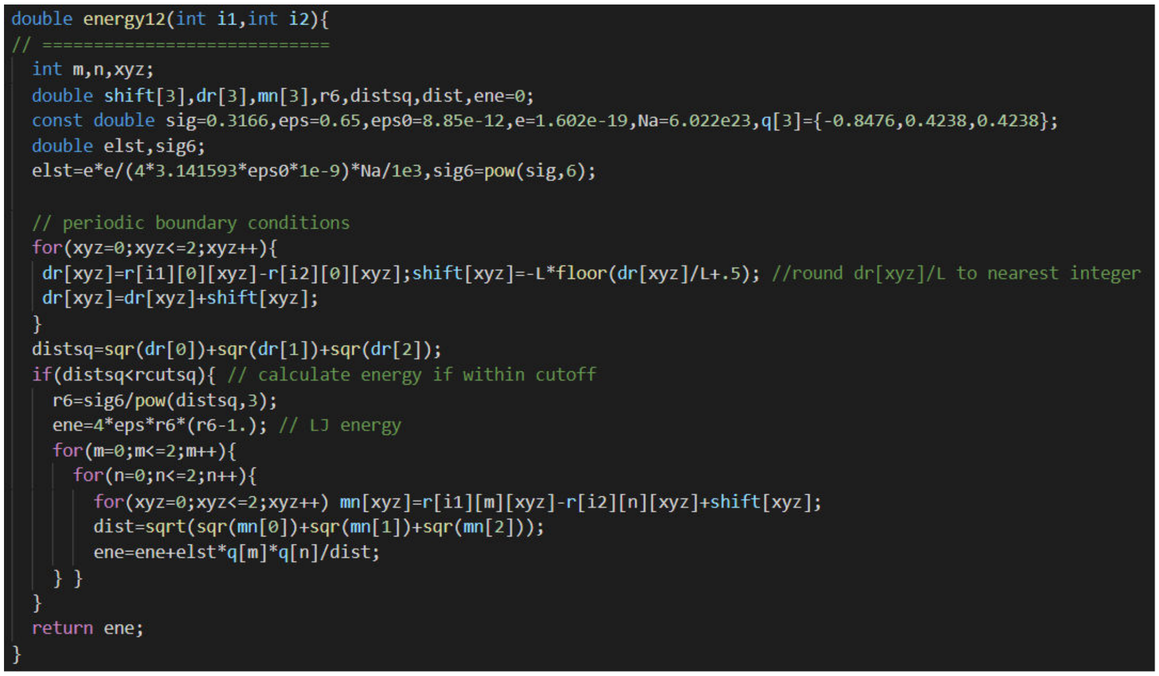
1. Reading of Data



```
printf("Program to calculate energy of water\n");
printf("Input NAME of configuration file\n");
scanf("%s",name); // reading of filename from keyboard
cputime0 = clock();    // assign initial CPU time (IN CPU CLOCKS)
gettimeofday(&start, NULL); // returns structure with time in s and us (microseconds)
fp=fopen(name, "r"); //opening of file and beginning of reading from HDD
fgets(line, LENGTH,fp); //skip first line
fgets(line, LENGTH,fp); sscanf(line,"%i",&natoms);
nmol=natoms/3; printf("Number of molecules %i\n",nmol);

for (i=0;i<nmol;i++){
  for(j=0;j<=2;j++){
    fgets(line, LENGTH,fp);
    sscanf(line, "%s %s %s %lf %lf %lf",nothing,nothing,nothing, &r[i][j][0],&r[i][j][1],&r[i][j][2]);
} }
printf("first line %lf %lf %lf\n",r[0][0][0],r[0][0][1],r[0][0][2]);
fscanf(fp, "%lf",&L); // read box size
printf("Box size %lf\n",L);
```

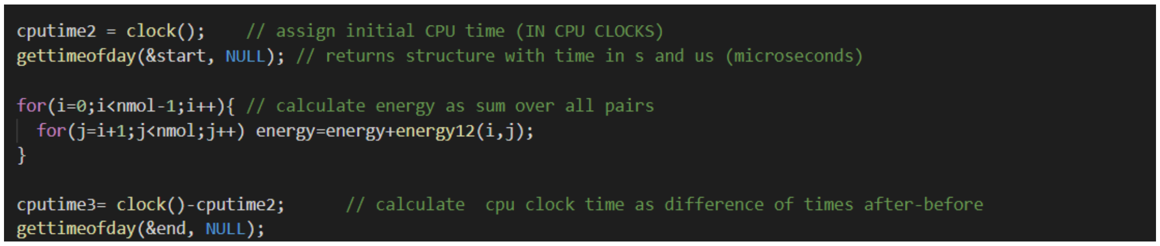
Serial code for reading the Data.

2. Energy of Configuration Calculation



```
double energy12(int i1,int i2){
// ============================
  int m,n,xyz;
  double shift[3],dr[3],mn[3],r6,distsq,dist,ene=0;
  const double sig=0.3166,eps=0.65,eps0=8.85e-12,e=1.602e-19,Na=6.022e23,q[3]={-0.8476,0.4238,0.4238};
  double elst,sig6;
  elst=e*e/(4*3.141593*eps0*1e-9)*Na/1e3,sig6=pow(sig,6);

  // periodic boundary conditions
  for(xyz=0;xyz<=2;xyz++){
   dr[xyz]=r[i1][0][xyz]-r[i2][0][xyz];shift[xyz]=-L*floor(dr[xyz]/L+.5); //round dr[xyz]/L to nearest integer
   dr[xyz]=dr[xyz]+shift[xyz];
  }
  distsq=sqr(dr[0])+sqr(dr[1])+sqr(dr[2]);
  if(distsq<rcutsq){ // calculate energy if within cutoff
    r6=sig6/pow(distsq,3);
    ene=4*eps*r6*(r6-1.); // LJ energy
    for(m=0;m<=2;m++){
      for(n=0;n<=2;n++){
        for(xyz=0;xyz<=2;xyz++) mn[xyz]=r[i1][m][xyz]-r[i2][n][xyz]+shift[xyz];
        dist=sqrt(sqr(mn[0])+sqr(mn[1])+sqr(mn[2]));
        ene=ene+elst*q[m]*q[n]/dist;
    } }
  }
  return ene;
}
```

Function to Calculate Energy of Configuration



```
cputime2 = clock();    // assign initial CPU time (IN CPU CLOCKS)
gettimeofday(&start, NULL); // returns structure with time in s and us (microseconds)

for(i=0;i<nmol-1;i++){ // calculate energy as sum over all pairs
  for(j=i+1;j<nmol;j++) energy=energy+energy12(i,j);
}

cputime3= clock()-cputime2;      // calculate  cpu clock time as difference of times after-before
gettimeofday(&end, NULL);
```

Code in main function to calculate the Energy wherein the outer loop needs to be parallelized

<u>Essential variables for energy calculation:</u>

nmol – number of molecules, determined from natoms

r[maxnum][3][3] – coordinates of atoms

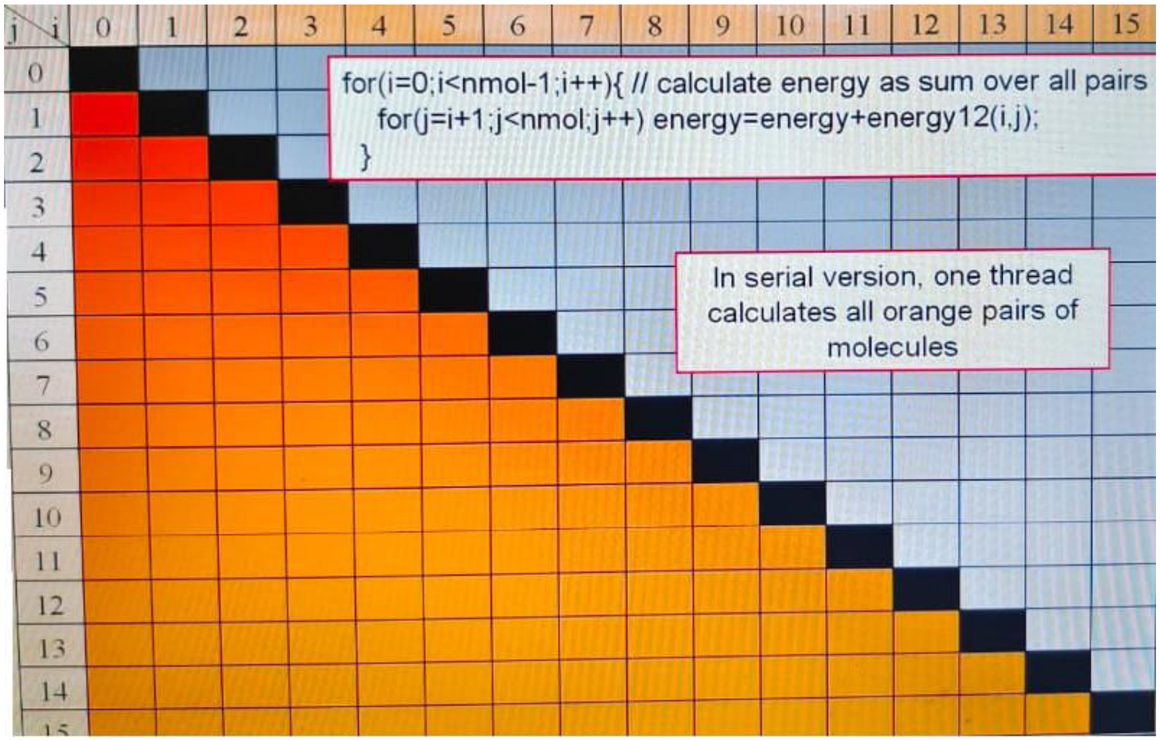
L – size of cubic box with configuration

Other parameters of calculation are declared as constants or derived from them

<u>Understanding the problem statement using 16 molecules:</u>

1. Serial Code calculates energy by forming systematic pairs (avoiding the self and repeated interactions)

#   Our job is to find a way to distribute the calculations in the orange triangles efficiently among the number of processes to achieve minimal load imbalance.

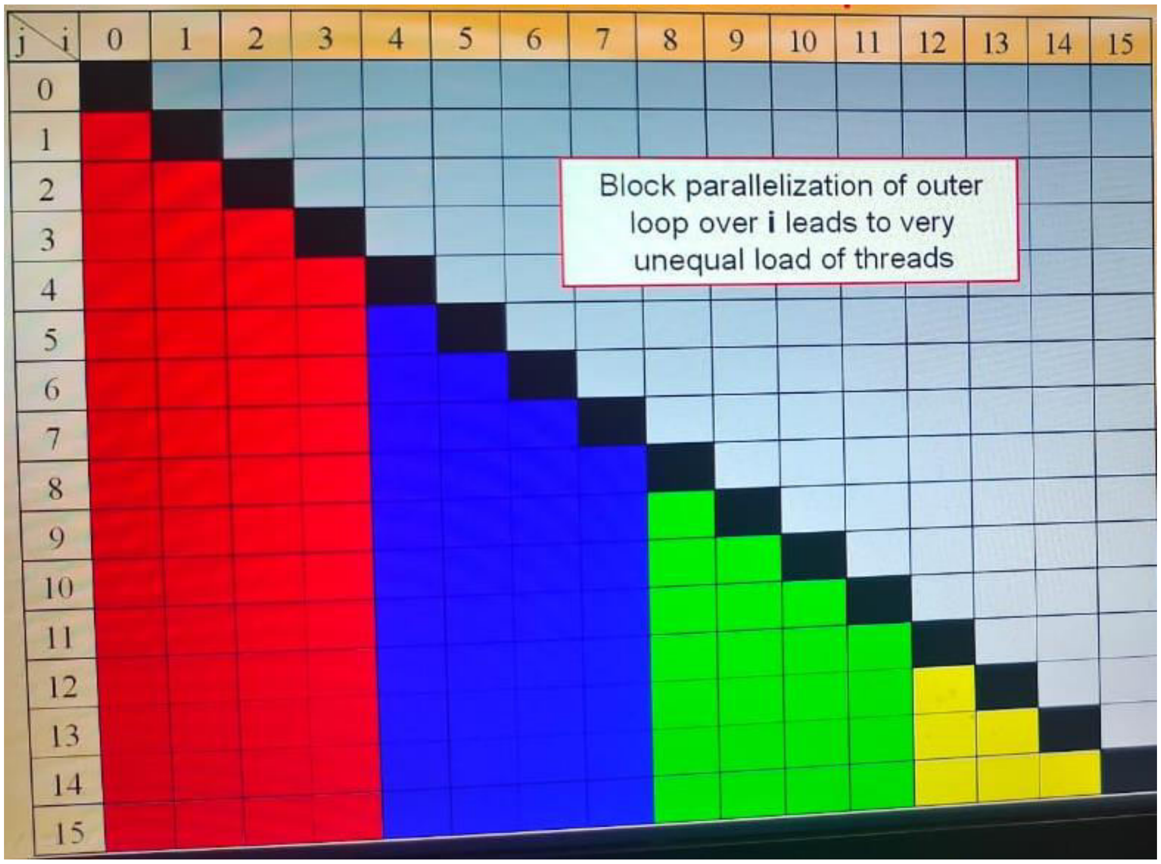


```
for(i=0;i<nmol-1;i++){ // calculate energy as sum over all pairs
   for(j=i+1;j<nmol;j++) energy=energy+energy12(i,j);
 }
```

In serial version, one thread calculates all orange pairs of molecules

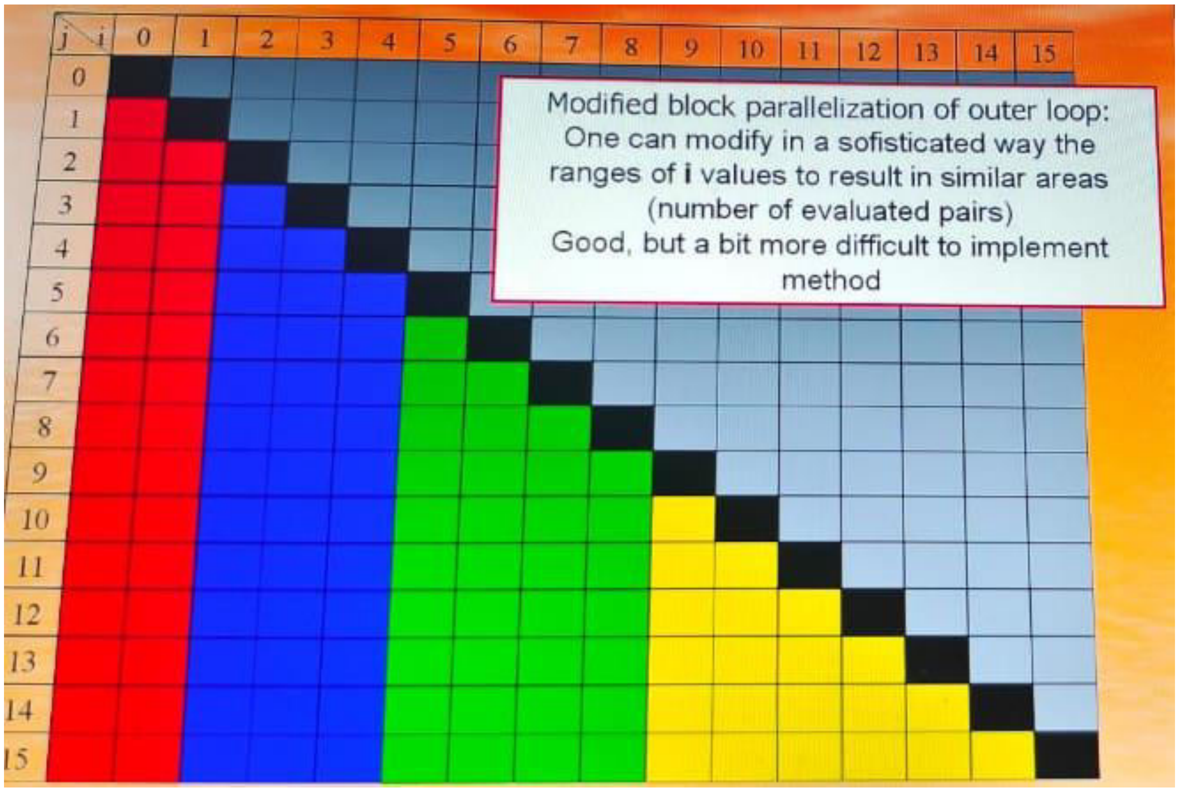
2. Approach 1: Bad Parallelization

# Distributing the **i** equally among all the processes. This kind of parallelization leads to bad values for load imbalance.


j
i
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Block parallelization of outer loop over **i** leads to very unequal load of threads

Approach 2: Good Parallelization

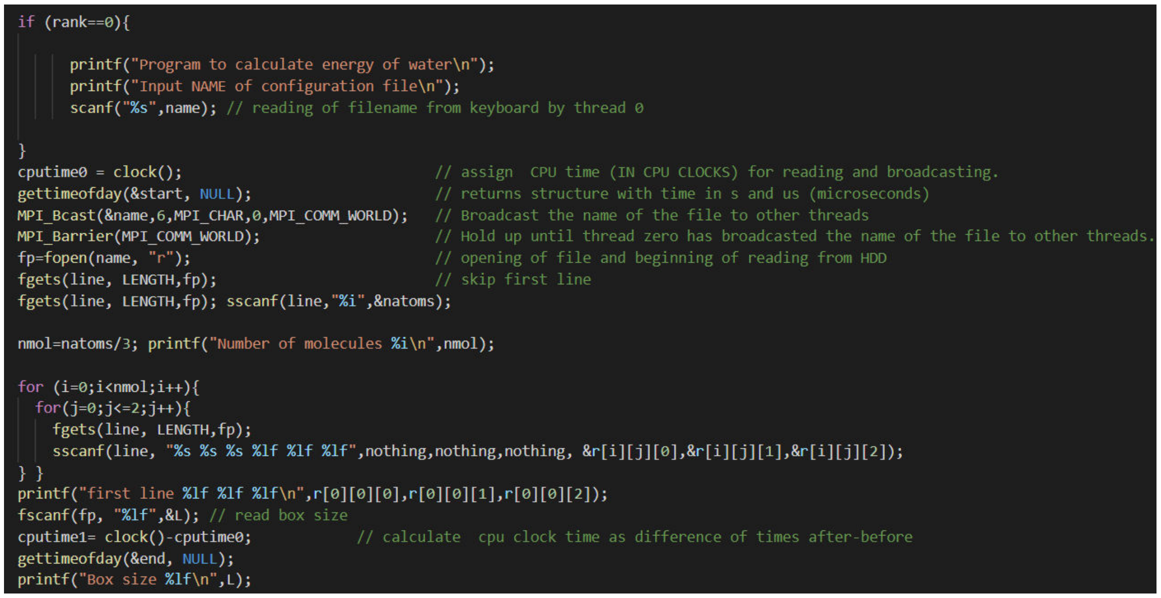
Here the problem is to distribute the area of triangle equally which are the number of calculations such that we achieve minimal load imbalance. **This project implements this method.**



Modified block parallelization of outer loop:
One can modify in a sofisticated way the ranges of **i** values to result in similar areas (number of evaluated pairs)
Good, but a bit more difficult to implement method

Objective 1 : **<u>To parallelize the reading of data using different settings and comparing the time differences.</u>**

Option 1 : One thread receives the file name and broadcasts it to other threads. No parallel modification, each thread opens the file individually serially for reading.



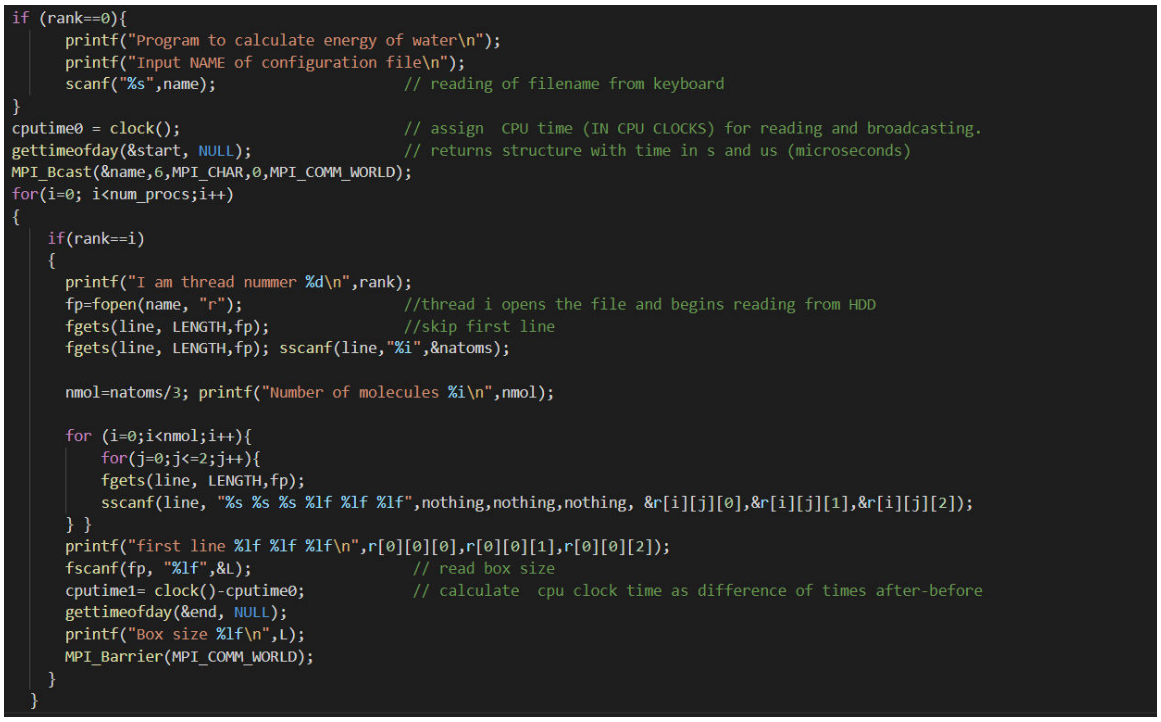
```
if (rank==0){

        printf("Program to calculate energy of water\n");
        printf("Input NAME of configuration file\n");
        scanf("%s",name); // reading of filename from keyboard by thread 0

}
cputime0 = clock();                             // assign  CPU time (IN CPU CLOCKS) for reading and broadcasting.
gettimeofday(&start, NULL);                     // returns structure with time in s and us (microseconds)
MPI_Bcast(&name,6,MPI_CHAR,0,MPI_COMM_WORLD);   // Broadcast the name of the file to other threads
MPI_Barrier(MPI_COMM_WORLD);                    // Hold up until thread zero has broadcasted the name of the file to other threads.
fp=fopen(name, "r");                            // opening of file and beginning of reading from HDD
fgets(line, LENGTH,fp);                         // skip first line
fgets(line, LENGTH,fp); sscanf(line,"%i",&natoms);

nmol=natoms/3; printf("Number of molecules %i\n",nmol);

for (i=0;i<nmol;i++){
  for(j=0;j<=2;j++){
    fgets(line, LENGTH,fp);
    sscanf(line, "%s %s %s %lf %lf %lf",nothing,nothing,nothing, &r[i][j][0],&r[i][j][1],&r[i][j][2]);
} }
printf("first line %lf %lf %lf\n",r[0][0][0],r[0][0][1],r[0][0][2]);
fscanf(fp, "%lf",&L); // read box size
cputime1= clock()-cputime0;             // calculate  cpu clock time as difference of times after-before
gettimeofday(&end, NULL);
printf("Box size %lf\n",L);
```

Limitations :

a. Unreliable transferability of programs to other OS/HW (Different limitations in number of opened files or concurrent accesses to a file)

b. Simultaneous reading of big files (>100 MB) results in significant reduction of access speed due to limited buffer of HDD and jumping of reading heads.

Option 2 : One thread receives the file name and broadcasts it to other threads. Reading of the file is not done simultaneously but sequentially controlled by a for loop.



```
if (rank==0){
      printf("Program to calculate energy of water\n");
      printf("Input NAME of configuration file\n");
      scanf("%s",name);                     // reading of filename from keyboard
}
cputime0 = clock();                         // assign  CPU time (IN CPU CLOCKS) for reading and broadcasting.
gettimeofday(&start, NULL);                 // returns structure with time in s and us (microseconds)
MPI_Bcast(&name,6,MPI_CHAR,0,MPI_COMM_WORLD);
for(i=0; i<num_procs;i++)
{
    if(rank==i)
    {
      printf("I am thread nummer %d\n",rank);
      fp=fopen(name, "r");                  //thread i opens the file and begins reading from HDD
      fgets(line, LENGTH,fp);               //skip first line
      fgets(line, LENGTH,fp); sscanf(line,"%i",&natoms);

      nmol=natoms/3; printf("Number of molecules %i\n",nmol);

      for (i=0;i<nmol;i++){
          for(j=0;j<=2;j++){
          fgets(line, LENGTH,fp);
          sscanf(line, "%s %s %s %lf %lf %lf",nothing,nothing,nothing, &r[i][j][0],&r[i][j][1],&r[i][j][2]);
      } }
      printf("first line %lf %lf %lf\n",r[0][0][0],r[0][0][1],r[0][0][2]);
      fscanf(fp, "%lf",&L);                  // read box size
      cputime1= clock()-cputime0;            // calculate  cpu clock time as difference of times after-before
      gettimeofday(&end, NULL);
      printf("Box size %lf\n",L);
      MPI_Barrier(MPI_COMM_WORLD);
    }
  }
```

Advantages:

a. File is accessed by a single process at a time.

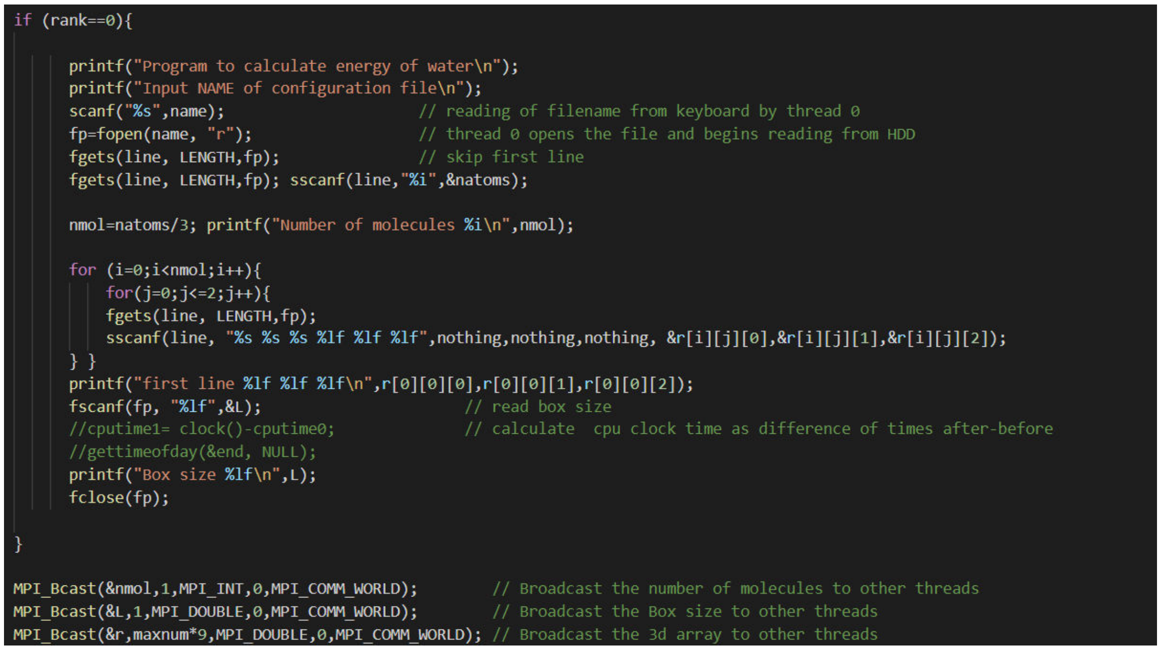
b. Reliable for any number of threads

Disadvantages:

a. Time of reading (and HDD load) is proportional to the number of processes.

b. Higher wear of HDD

Option 3 (Recommended) : Data is read by a single process and then it is broadcasted to all the other processes. Here only the root thread have to worry about reading the data and the others will begin their calculations as soon as they receive it.



```c
if (rank==0){

    printf("Program to calculate energy of water\n");
    printf("Input NAME of configuration file\n");
    scanf("%s",name);                    // reading of filename from keyboard by thread 0
    fp=fopen(name, "r");                 // thread 0 opens the file and begins reading from HDD
    fgets(line, LENGTH,fp);              // skip first line
    fgets(line, LENGTH,fp); sscanf(line,"%i",&natoms);

    nmol=natoms/3; printf("Number of molecules %i\n",nmol);

    for (i=0;i<nmol;i++){
        for(j=0;j<=2;j++){
        fgets(line, LENGTH,fp);
        sscanf(line, "%s %s %s %lf %lf %lf",nothing,nothing,nothing, &r[i][j][0],&r[i][j][1],&r[i][j][2]);
    } }
    printf("first line %lf %lf %lf\n",r[0][0][0],r[0][0][1],r[0][0][2]);
    fscanf(fp, "%lf",&L);                     // read box size
    //cputime1= clock()-cputime0;             // calculate  cpu clock time as difference of times after-before
    //gettimeofday(&end, NULL);
    printf("Box size %lf\n",L);
    fclose(fp);

}

MPI_Bcast(&nmol,1,MPI_INT,0,MPI_COMM_WORLD);        // Broadcast the number of molecules to other threads
MPI_Bcast(&L,1,MPI_DOUBLE,0,MPI_COMM_WORLD);        // Broadcast the Box size to other threads
MPI_Bcast(&r,maxnum*9,MPI_DOUBLE,0,MPI_COMM_WORLD); // Broadcast the 3d array to other threads
```

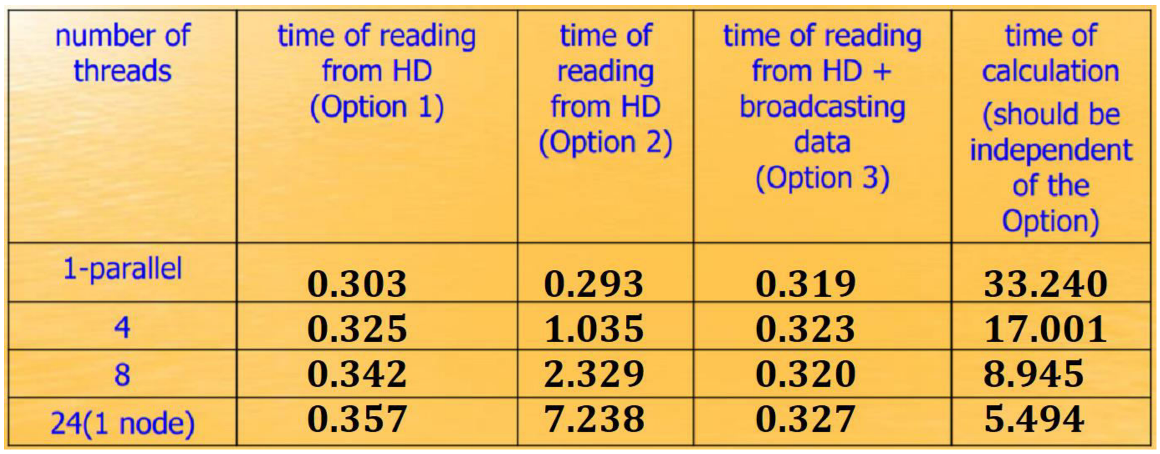
Advantages:

a. File is read only once by a single thread.

b. Minimal demand on HDD (as in serial version)

c. High speed of reading the data, independent of number of processes communication among processes during distribution of data can be significantly optimized and speeded up by SW implementation of MPI environment (in contrast to option 2 with no optimalization)

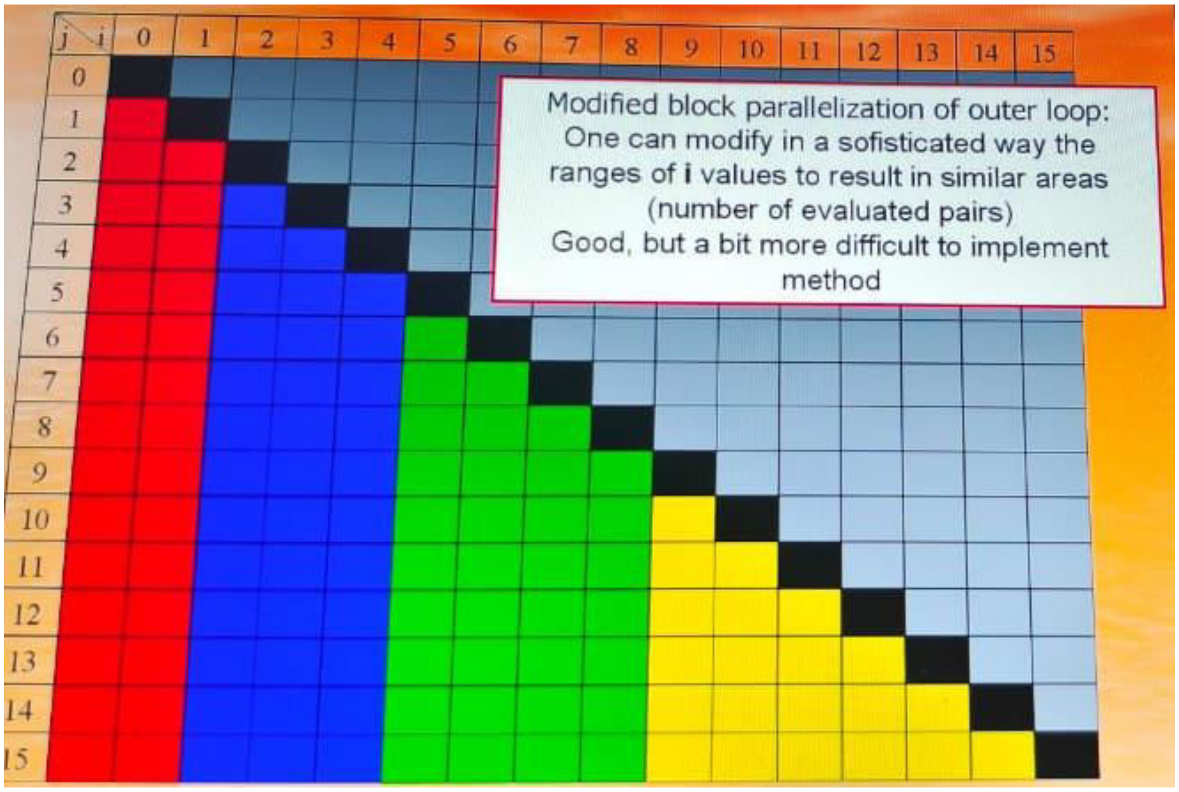
Performance Comparison with different reading options for the 100k.gro Configuration - Maximum walltimes across all the threads and calculation in seconds (obtained by MPI_Reduce with operation MPI_MAX).



number of threads	time of reading from HD (Option 1)	time of reading from HD (Option 2)	time of reading from HD + broadcasting data (Option 3)	time of calculation (should be independent of the Option)
1-parallel	0.303	0.293	0.319	33.240
4	0.325	1.035	0.323	17.001
8	0.342	2.329	0.320	8.945
24(1 node)	0.357	7.238	0.327	5.494

# Clearly the Option 3 is best for reading as evident from the table and we will be using it throughout this project.

Objective 2 : Understanding the method of Parallelization of the Calculations.



Modified block parallelization of outer loop:
One can modify in a sofisticated way the ranges of **i** values to result in similar areas (number of evaluated pairs)
Good, but a bit more difficult to implement method

1. As discussed above, energy will be calculated for the total number of unique pairs as seen above.

2. The problem statement can be thought out as parallelizing the total calculations in the triangle by diving the area of the triangle (aka the calculations) equally among the number of threads such that each thread recieves similar quantity of calculations to minimise our load imbalance.

3. Variables required :

```
long n = (long)((nmol)*(nmol-1))/(2*num_procs);
long *arr = decider(num_procs,nmol-1,n);
long *arr2 = decider2(num_procs,nmol-1,n);
```

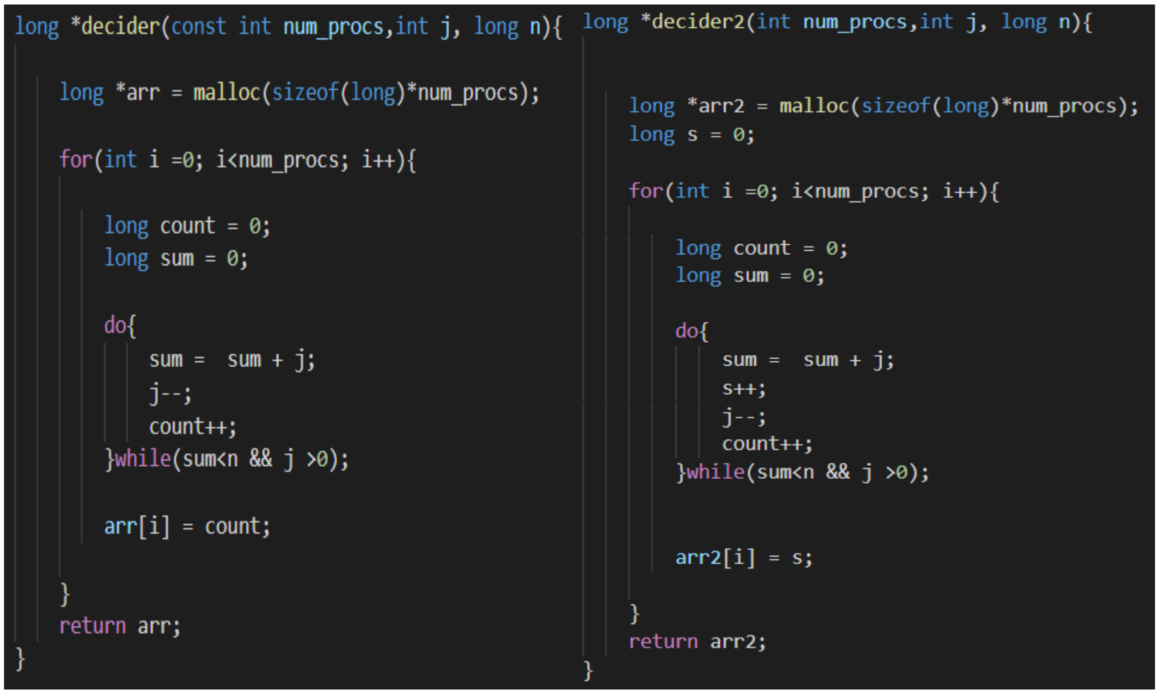
n : Equal Area of calculations to be received by each thread

array arr : Stores the steps travelled by each thread, eg in the above diagram, the first thread will receive the red area which has 29 steps in total.

array arr2 : Till which step a thread will perform calculation.

Therefore, starting point of each thread will be -> arr2[rank]-arr[rank]

The functions decider and decider2 just perform the internal calculations to achieve equal distribution of area. Refer the code synapses below.

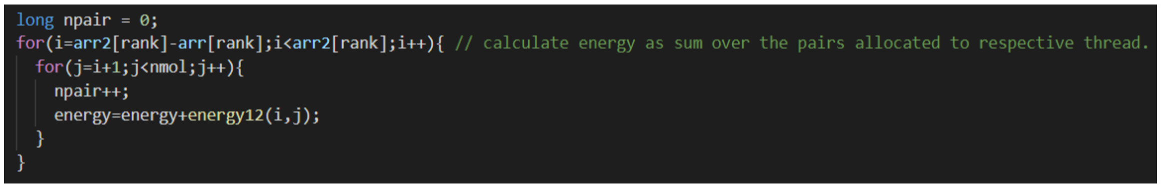


```
long *decider(const int num_procs,int j, long n){

    long *arr = malloc(sizeof(long)*num_procs);

    for(int i =0; i<num_procs; i++){

        long count = 0;
        long sum = 0;

        do{
            sum =  sum + j;
            j--;
            count++;
        }while(sum<n && j >0);

        arr[i] = count;

    }
    return arr;
}
```

```
long *decider2(int num_procs,int j, long n){

    long *arr2 = malloc(sizeof(long)*num_procs);
    long s = 0;

    for(int i =0; i<num_procs; i++){

        long count = 0;
        long sum = 0;

        do{
            sum =  sum + j;
            s++;
            j--;
            count++;
        }while(sum<n && j >0);


        arr2[i] = s;

    }
    return arr2;
}
```

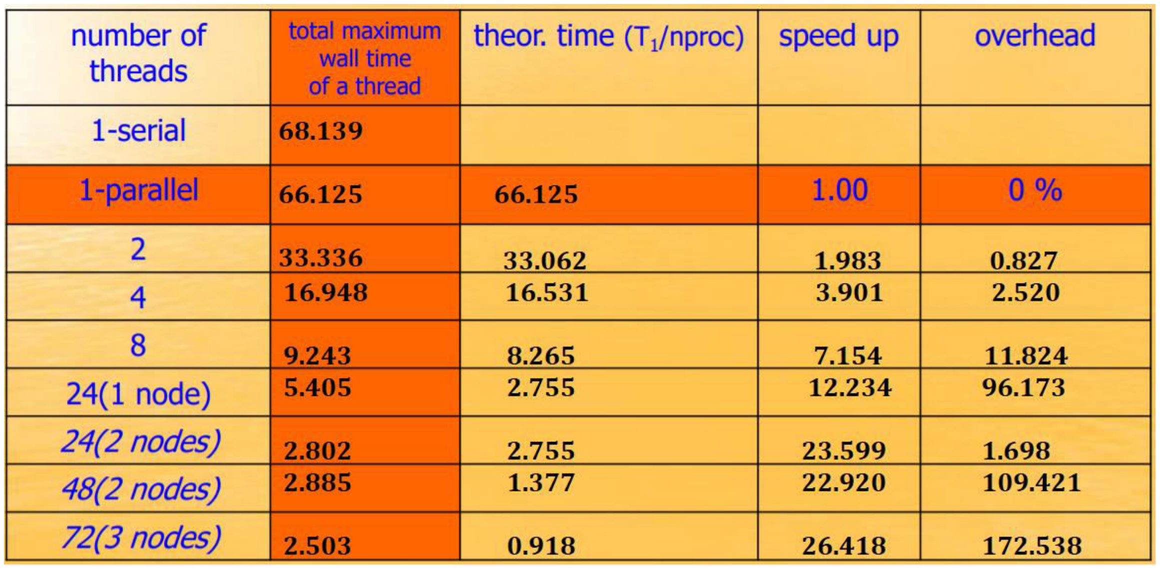
# Parallelizing the outer Loop :

It can be seen that the respective thread will pick up its own steps according to their rank which will be passed as an index in the respective arrays.



```
long npair = 0;
for(i=arr2[rank]-arr[rank];i<arr2[rank];i++){ // calculate energy as sum over the pairs allocated to respective thread.
  for(j=i+1;j<nmol;j++){
    npair++;
    energy=energy+energy12(i,j);
  }
}
```

After running the above parallel strategy for different processors and cpu settings for the **100k.gro** file, the results were as follows :

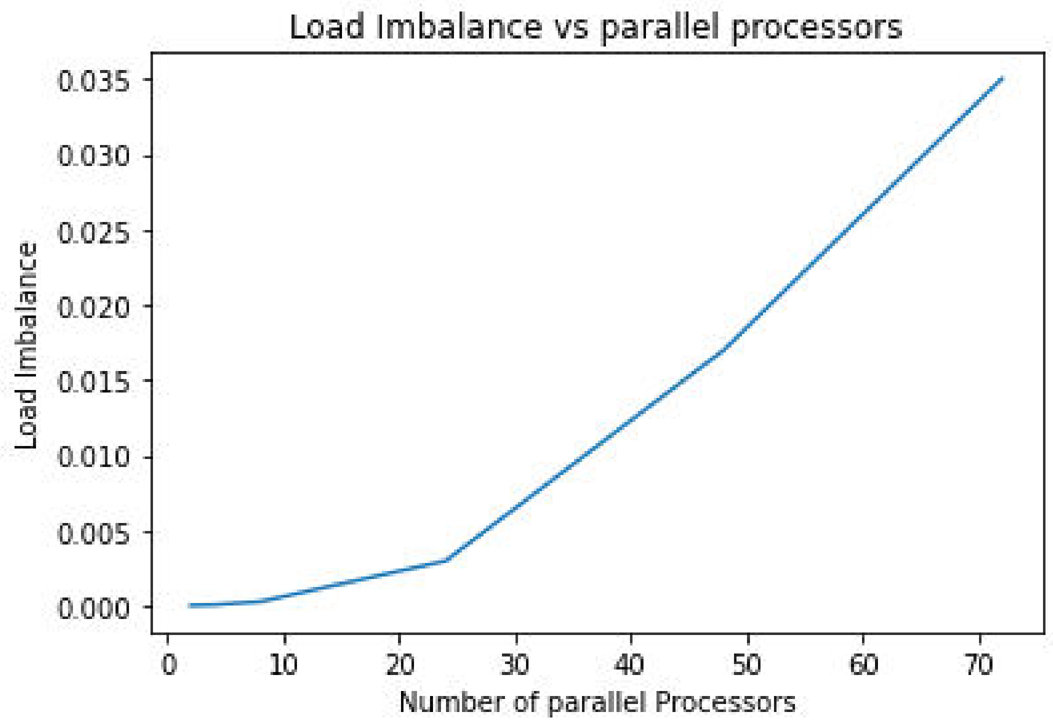


number of threads	total maximum wall time of a thread	theor. time (T_1/nproc)	speed up	overhead
1-serial	68.139			
1-parallel	66.125	66.125	1.00	0 %
2	33.336	33.062	1.983	0.827
4	16.948	16.531	3.901	2.520
8	9.243	8.265	7.154	11.824
24(1 node)	5.405	2.755	12.234	96.173
24(2 nodes)	2.802	2.755	23.599	1.698
48(2 nodes)	2.885	1.377	22.920	109.421
72(3 nodes)	2.503	0.918	26.418	172.538

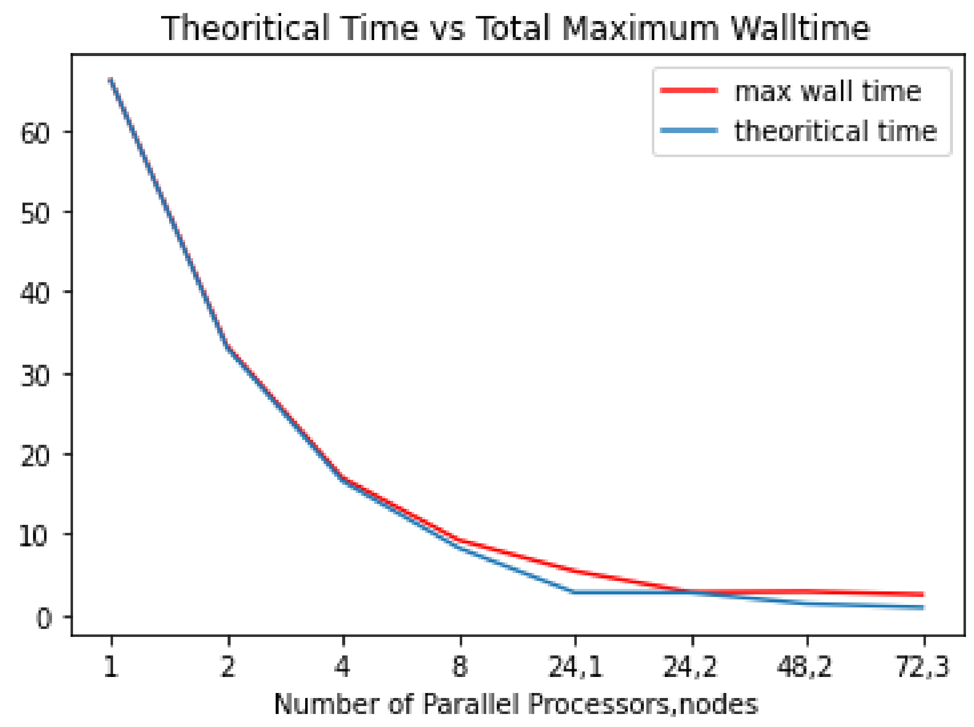
# **Observations:**

1. Load Imbalance :

It can be observed that as the number of parallel processors increases, the load imbalance also increases. Minimum Load Imbalance achieved for 2 parallel processors = **0.000047** and max load imbalance at 72 processors = **0.035**

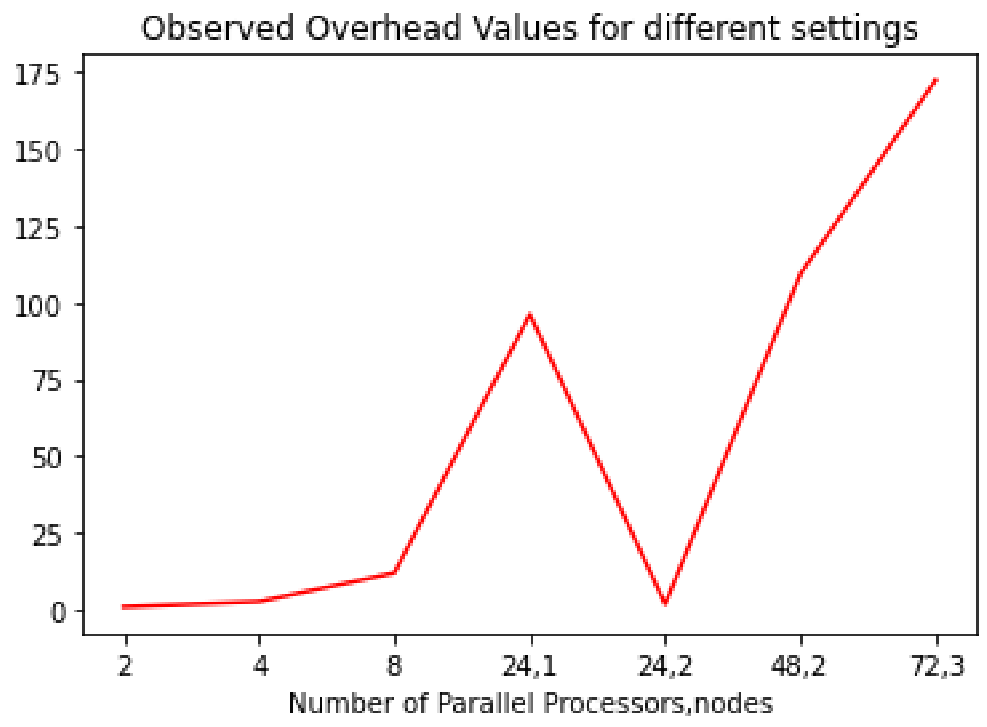


2. The total maximum wall time of a thread follows an inverse relationship with the number of threads but follows a similar trend to the theoritical time



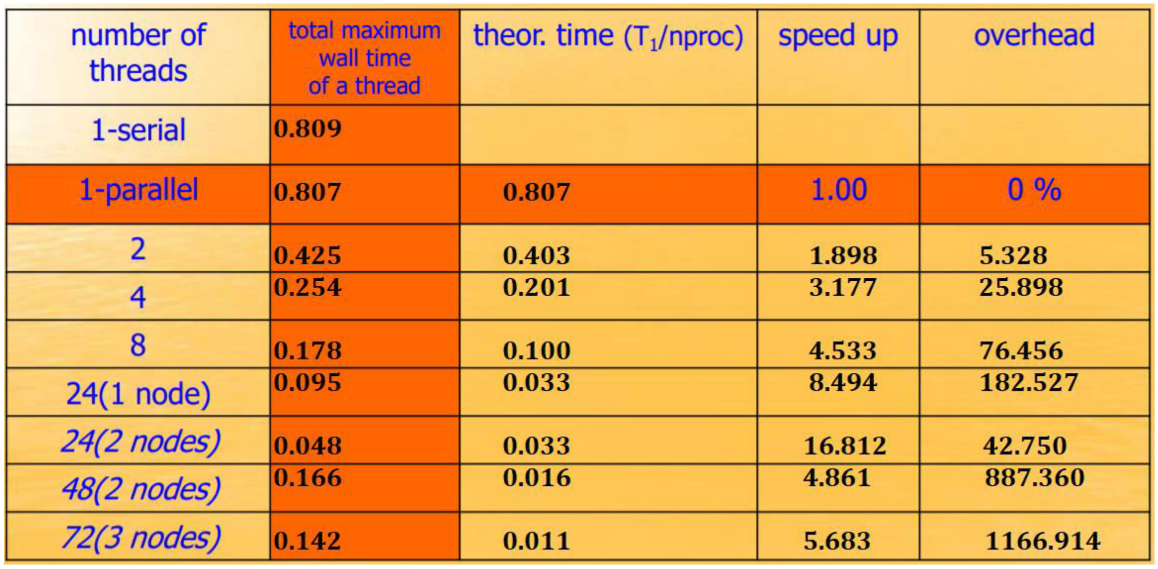
3. As evident from the table, **Speedup** follows a direct relationship with the number of parallel processors.

4. As evident from the graph the **overhead** decreases when multiple nodes are added, given that more nodes help in decreasing the wall time of a thread.



## Running the same experiment for 10k.gro file :

After running the above parallel strategy for different processors and cpu settings for the **10k.gro** file, the results were as follows and the observations are similar to that of the 100k.gro file :

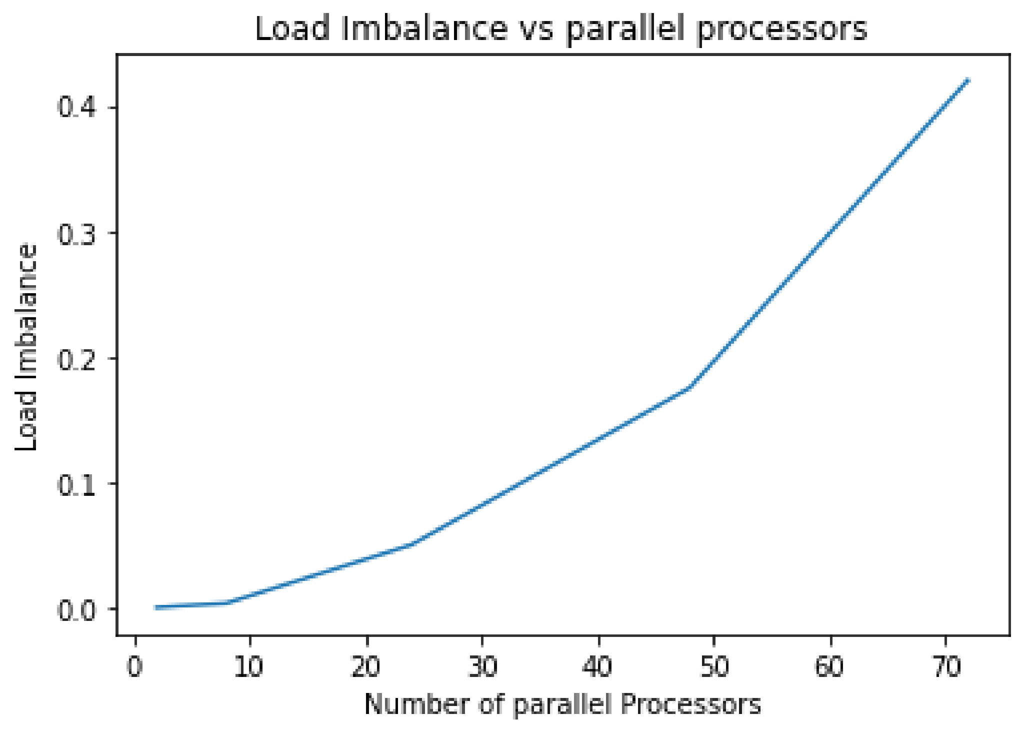


number of threads	total maximum wall time of a thread	theor. time (T_1/nproc)	speed up	overhead
1-serial	0.809			
1-parallel	0.807	0.807	1.00	0 %
2	0.425	0.403	1.898	5.328
4	0.254	0.201	3.177	25.898
8	0.178	0.100	4.533	76.456
24(1 node)	0.095	0.033	8.494	182.527
24(2 nodes)	0.048	0.033	16.812	42.750
48(2 nodes)	0.166	0.016	4.861	887.360
72(3 nodes)	0.142	0.011	5.683	1166.914

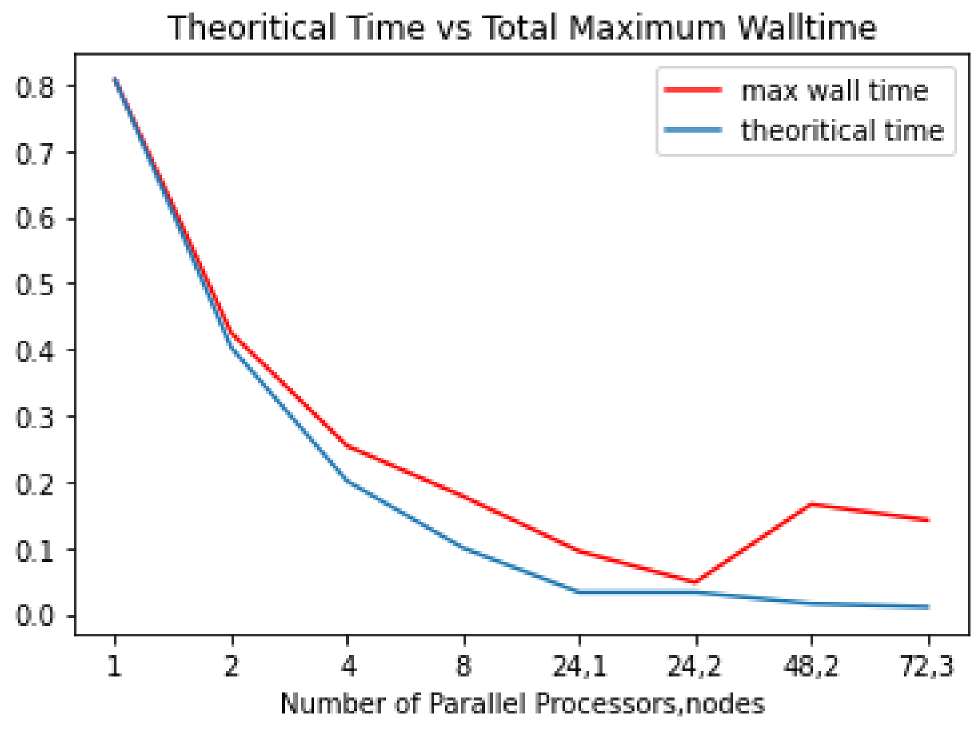
# Observations :

1. Load Imbalance:

It can be observed that as the number of parallel processors increases, the load imbalance also increases. Minimum Load Imbalance achieved for 2 parallel processors = **0.0001** and max load imbalance at 72 processors = **0.420**



2. The total maximum wall time of a thread follows an inverse relationship with the number of threads but follows a similar trend to the theoritical time with a nudge owing to the discrepancy caused by adding extra nodes.



3. As evident from the table, **Speedup** follows a direct relationship with the number of parallel processors.

4. As evident from the table the **overhead** decreases when multiple nodes are added, given that more nodes help in decreasing the wall time of a thread.

# IMP NOTE : Parallelization of a parallel algorithm beyond a certain point causes the program to run slower (take more time to run to completion). This is typically result of a communications bottleneck. One should keep it in check .