Multi-core and Multi-Process Parallel Programming in Molecular Dynamic Algorithm

Work Assignment 3

Carlos Sá - A59905
a59905@alunos.uminho.pt

Sérgio Caldas - A57779
a57779@alunos.uminho.pt

Filipe Oliveira - A57816
a57816@alunos.uminho.pt

June 26, 2016

Abstract

*This report is a result of a study about Molecular Dynamic Algorithm (MD). In this work MD simulation source code written in **C** is provided. The goal of this work is study MD program, his sequential algorithm complexity, and implement a multi-core parallel version in two different programming paradigms: Shared Memory and Distributed memory. The shared memory version of MD will be achieved using OpenMP - an API for multi-platform shared-memory parallel programming in C/C++ and Fortran. The distributed version of MD will be achieved using MPI (Message Passing Interface) using OpenMPI - an high performance message passing library. Initially, we'll study the program organization (with **gprof** and **callgrind**) and study his complexity. For all MD implementations we're going to produce a performance analysis using compute-641 node of SeARCH Cluster and draw some conclusions.*

I. Molecular Dynamic: Program code Analysis

The starting point of this work is the MD sequential program wrote in C. This program performs a simulation of molecular dynamic. The **main** file of this program is essentialy divided into 5 different phases:

1. Initialize MD structure (function **initialiseMD**);
2. Particles creation (function **createParticules**);
3. Particles initialization (function **initialiseParticles**);
4. Run the algorithm (function **runiters**);
5. Returned results validation (function **JGFvalidate**);

Structurally, MD program has 4 **C files** and the correspondent 4 **header files**. Another extra header file is used for datatypes of MD struct, and particles position, speed, and force. The following call-graph (figure 1) can easily give us an idea of the programs structure and calling relationship between functions:

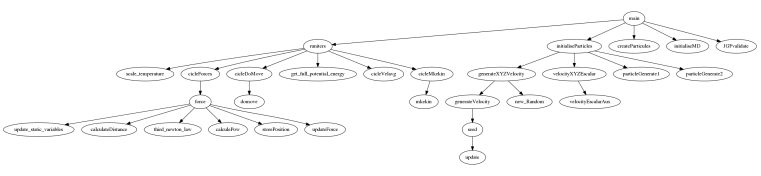

Figure 1: *MD Program Structure*

We should perform some tests in order to understand what functions we should focus on to improve their performance.

Before running the algorithm, we need to consider two values:

- **md->side**: the side;
- **md->rcoff**: the cutter radius measure;

For the entire analysis we're going to produce in this work, we'll not change this two values. By default, **md->side** considered is 21.946755 and **md->rcoff** is 3.25.

This program has different sizes to simulate the program with a different number of particles. In this work, we're going to use **size 1** wich runs the simulation for **8788** particles and **size 3** for **19652** particles. We choose this datasets because **size 1** is the default size and **size 3** uses more than two times more particles.

After using **gprof** we made a **flat profile** and a **callgraph profile** in order to achieve the performance critical path. After doing this we get the following result from **gprof** profile:

```
Flat profile:

Each sample counts as 0.01 seconds.
      cumulative   self              self     total
time   seconds    seconds    calls  us/call  us/call     name
99.89    27.44     27.44   439400    62.45    62.45      force
0.04     27.45     0.01    439400     0.02     0.02      domove
0.04     27.46     0.01    439400     0.02     0.02      mkekin
0.04     27.47     0.01                                 runiters
0.00     27.47     0.00    13182      0.00     0.00      seed
0.00     27.47     0.00       50      0.00     0.00      scale_temperature
0.00     27.47     0.00        1      0.00     0.00      new_Random
0.00     27.47     0.00        1      0.00     0.00      particleGenerate2
0.00     27.47     0.00        1      0.00     0.00      velocityXYZEscalar
```

We can see that **force**, **domove** and **mkekin** are the 3 most invoqued functions with a total of 439400 calls each. The function **force** is called inside **cicleForces**, **domove** is called inside **cicleDoMove**, and **mkekin** is called inside **cicleMkekin** function. All of them are *void* functions called inside **runiters** on **MD.c file** wich is called from main (see left side of figure 1).

Particularly, **force** function is consuming **99.89%** of the total execution time. In order to secure this conclusion, we performed a second analysis with **callgrind** - a profilling tool integrated in valgrind. After using **gprof2dot**[1] (an additional tool to generate a visual representation of callgraph from the callgrind's collected data) we could get the following critical path:

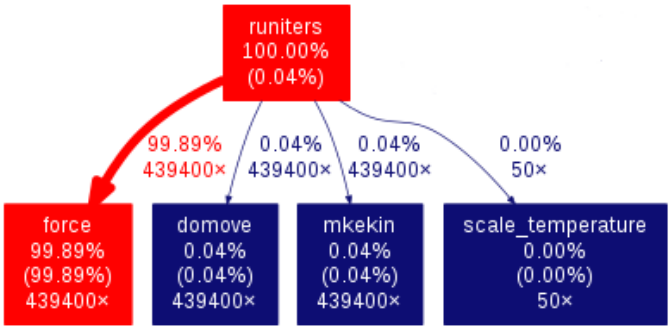

Figure 2: *Critical path of MD program in red.*

At this point, we know now that **force**, **domove**, and **mkekin** are the 3 most invoqued functions. We can also assure now without any doubt, that our study focus should be on **force** function from the provided code. Since this is the heaviest function in terms of performance, if we could optimize and made a good paralelization on it, we should expect a significant improvement in speedup.

From the 30 functions (figure 1) of MD program, we could identify the 3 most called functions and the heaviest function in terms of performance. In the next section we're going to look at these 3 functions and study their complexity.

II. Sequential MD Version Asymptotic Analysis

In the previous section we made a profile on the provided code. From the 30 functions of MD code simulation, we achieved the 3 most called functions (force, domove and mkekin) and the heaviest one in terms of performance (force). These three functions are all called **439400 times**. In the next sections, we're going to study a paralelization with **OpenMP** and **MPI** so these three functions should be considered in that analysis.

For now, if we want to reduce the execution time, we definitely pay atention and study the **force** function. For computing algorithm complexity, we need to identify how many times the intermediate for cicles was runned as complexity of the intermediate functions from main, to force.

In the next function codes, we can see the relevant function calls from **main** to **force** marked in red. The total executions of *for* cicles and intermediate functions are marked in orange. We are going to use **big-O notation** for computing the complexity.

Remember the figure 1. The **main** calls **runiters** function **one time**. The function **runiters** code is ilustrated below:

basicstylebasicstyle

```
void runiters(MD *md, Particles *particulas){

 printf("Side:%f RaiodeCorte: %f\n",md->side,md->rcoff);

    for (md->move = 0; md->move < md->movemx; md->move++) {
       cicleDoMove (md,particulas);
       cicleForces (md,particulas);
       cicleMkekin (md,particulas);
       cicleVelavg (md,particulas);
       (.........)
    }
}
```

The function **cicleForces** compute the particles forces and his code is presented below:

basicstylebasicstyle

```
void cicleForces(MD *md,Particles *particulas){
    int i;
    md->epot = md->vir = 0.0;

    for (i = 0; i < md->mdsize; i++)
        force(md,particulas,i);
}
```

and finally, after calling **cicleForces** **md->movemx** times, we call **force** function **md->mdsize** times wich code we can see below:

basicstylebasicstyle

```
void force(MD *md,Particles *particulas, int pos){

    (.....irrelevant declarations.....)

    storePosition(posActPart,particulas,pos);
    for (i = pos + 1; i < md ->mdsize; i++) {
      powDist[0] = calculateDistance(dist,posActPart,particulas,i,sideh,md);
      if(powDist[0] <= rcoffs) {
        calculePow(powDist);
        leiNewton[0] = dist[0] * powDist[7];
        force[0] *= leiNewton[0];
        leiNewton[1] = dist[1] * powDist[7];
        force[1] *= leiNewton[1];
        leiNewton[2] = dist[2] * powDist[7];
        force[2] *= leiNewton[2];
        third_newton_law(particulas,i, leiNewton);
        update_static_variables(powDist, md);
      }
    }
    updateForce(particulas,force,pos);
}
```

In order to compute algorithm complexity correctly, we need to know the values for **md->movemx**. So, after inspecting **MD.c** file we can see that **md->movemx** value is **50**. So we're going to consider that **md->movemx** can be represented by ***k***. The **md->mdsize** is the size of our input.

To our *asymptotic analysis* we'll use ***n* to represent md->mdsize** and we'll consider this ***n*** is bigger. Inside **force**, we'll also need to contabilize the complexity of **calculePow**, **third_newton_law**, **update_static_variables** and **updateForce**. All of these functions just perform memory access and has no loop so we'll consider they execute $k \times n \times n$ times. For those functions we achieved the complexity $\mathcal{O}(n^2)$. For **force** we can achieve the following formulation to determine the complexity:

[1] gprof2dot github repository here: `https://github.com/jrfonseca/gprof2dot`

$$\sum_{i=0}^{md->movemx} \left(\sum_{j=0}^{md->mdsize} \left(\sum_{k=j+1}^{md->mdsize} 1 \right) \right)$$

we'll use instead:

$$\sum_{i=0}^{k} \left(\sum_{j=0}^{n} \left(\sum_{k=j+1}^{n} 1 \right) \right)$$

And making all intermediate computations we can say this algorithm executes in $\mathcal{O}(n^2)$. Algorithms in $\mathcal{O}(n^2)$ are generally known for being good algorithms to parallelize. Whenever n doubles, the running time increases fourfold. The main idea of this kind of algorithms is simulating the interaction of all particles. Each particle interacts with all others, so we have a $\mathcal{O}(n^2)$ complexity. Paralelization of MD program will be explored in OpenMP MD's implementation (section V) and MPI MD's implementation (section VI).

III. The Best Sequential version of MD

While we're doing the program analysis and determine algorithm complexity, we could identify that **force** function is the heaviest function and one of the most called function. We know that a fair paralelization is the one who is implemented over the best sequential version we can achieve. So we looked to the code of **force** function (and all functions invoqued in it), and manually tried to perform possible optimizations. For example, lets take a look into **force** function again below. Please notice the red parts:

basicstylebasicstyle

```
void force(MD *md,Particles *particulas, int pos){

    (.....irrelevant declarations.....)

    storePosition(posActPart,particulas,pos);
    for (i = pos + 1; i < md ->mdsize; i++) {
      powDist[0] = calculateDistance(dist,posActPart,particulas,i,sideh,md);
      if(powDist[0] <= rcoffs) {
        calculePow(powDist);
        leiNewton[0] = dist[0] * powDist[7];
        force[0] *= leiNewton[0];
        leiNewton[1] = dist[1] * powDist[7];
        force[1] *= leiNewton[1];
        leiNewton[2] = dist[2] * powDist[7];
        force[2] *= leiNewton[2];
        third_newton_law(particulas,i, leiNewton);
        update_static_variables(powDist, md);
      }
    }
    updateForce(particulas,force,pos);
}
```

For each time the **if** condition in orange is verified, we execute calculatePow function one time, and access powDist[7] more 3 times. This can happen repeatedly at a maximum of **md->mdsize** times because the *if* condition is inside the *for* loop. We can convert this 3 memory accesses of powDist[7] in just 1 by doing the following modifications:

basicstylebasicstyle

```
void force(MD *md,Particles *particulas, int pos){

    (.....irrelevant declarations.....)
    int aux;

    storePosition(posActPart,particulas,pos);
    for (i = pos + 1; i < md ->mdsize; i++) {
      powDist[0] = calculateDistance(dist,posActPart,particulas,i,sideh,md);
      if(powDist[0] <= rcoffs) {
        aux = powDist[7];
        calculePow(powDist);
        leiNewton[0] = dist[0] * aux;
        force[0] *= leiNewton[0];
        leiNewton[1] = dist[1] * aux;
        force[1] *= leiNewton[1];
        leiNewton[2] = dist[2] * aux;
        force[2] *= leiNewton[2];
        third_newton_law(particulas,i, leiNewton);
        update_static_variables(powDist, md);
      }
    }
    updateForce(particulas,force,pos);
}
```

Remember that force function is called **md->mdsize** times. If we consider a huge value for **md->mdsize**, we thought that we can reduce a significant number of memory accesses if **powDist** array doesn't fit in the cache. We can use the same technique in calculatePow since its inside the for loop too by converting the original code:

basicstylebasicstyle

```
void calculePow(double powDist[]){
  powDist[1] = 1.0/powDist[0];

  powDist[2] = powDist[1] * powDist[1];
  powDist[3] = powDist[2] * powDist[1];
  powDist[4] = powDist[2] * powDist[2];
  powDist[6] = powDist[2] * powDist[4];
  powDist[7] = powDist[6] * powDist[1];
  powDist[8] = powDist[7] - 0.5 * powDist[4];
}
```

into this version:

basicstylebasicstyle

```
void calculePow(double powDist[]){
  powDist[1] = 1.0/powDist[0];
  int aux1 = powDist[1];
  powDist[2] = aux1 * aux1;
  int aux2 = powDist[2];
  powDist[3] = aux2 * aux1;
  powDist[4] = aux2 * aux2;
  int aux4 = powDist[4];
  powDist[6] = aux2 * aux4;
  powDist[7] = powDist[6] * aux1;
  powDist[8] = powDist[7] - 0.5 * aux4;
}
```

By doing this we access:

- powDist[1] 2 times instead of 5 (with aux1);
- powDist[2] 2 times instead of 5 (with aux2);
- powDist[4] 2 times instead of 3 (with aux4);

After performing this modifications, we noticed that **powDist** just actually holds 9 doubles. Since it fits on the cache of 641 node (2) we can't expect relevant speedup improvement. But these modifications are good practices since accessing memory data in some case studies can actually be the cause of performance degradation.

We used **perf** for accessing some performance counters just to find out if there is any impact of the technique we applied here. We measure the available counters on **641** node (2) with the original sequential code, and with our modified version described above using **aux** variables. We've summarize all the results on table 1:

EVENT	Original_MD	Modified_MD
cpu-cycles	37 689 379 124	26 569 769 211
instructions	45 626 052 978	45 449 012 947
cache-references	118 324 469	59 834 842
cache-misses	746 357	239 645
branch-instructions	11 272 336 357	11 194 114 369
branch-misses	306 084 742	390 077 541
bus-cycles	1 453 448 232	1 023 957 898
cpu-clock (msec)	105 105	73 624
task-clock (msec)	105 092	73 616
page-faults	366	594
context-switches	3743	2019
cpu-migrations	2	2
minor-faults	509	376

Table 1: *Results of relevant performance counters using* ***perf***

Both codes were compiled with same version of the **C** compiler (**gcc**) and aggressive optimization flags (**-O3**). More info about compilation in **work methodology** (section IV). The green results reflects the improvements we achieved with our MD sequential modified by the workgroup.

The modifications described above were the only modifications performed by the workgroup in the algorithm. This modifications doesn't change the complexity of the algorithm so the final sequential version still executes in $\mathcal{O}(n^2)$. This is the best sequential version we achieved of MD program, and that's the version we'll gonna use for paralelization.

IV. Working Methodology

In order to analize this algorithm, the group made the tests, for a sequencial and parallel (shared memory, distributed memory) kernel's, in one cluster (search6) node. The chosen node was the node compute-641. The specifications of this node can be consulted in the table 2.

System	compute-641 node
# CPUs	2
CPU	Intel® Xeon® E5-2650v2
Architecture	Ivy Bridge
# Cores per CPU	8
# Threads per CPU	16
Clock Freq.	2.6 GHz
L1 Cache	256 KB 32 KB por core
L2 Cache	2048 KB 256 KB por core
L3 Cache	20 MB
Inst. Set Extensions	AVX
#Memory Channels	4
Memory BW	59.7 GB/s

Table 2: *The compute-641 processor's characterization*

The three kernel's was compiled with the GNU compiler (gcc) version 4.9.0. After the programs compilations we run a script that measure the execution time. The measure methodology used, was the K-Best for a K=5 samples. For this 5 samples we chosen the best one. In the OpenMP MD version we've followed the same approach. We tested the program for a multi sets of threads. For each set, we collect 5 samples and selected the best one too.

Between two measurements, we force the kernel to sleep 3 seconds for stabilize the measurement by each two executions.

On the MPI implementation we used **GNU 4.9.3**, **OpenMPI 1.8.4** for results using **ethernet** and **myrinet** net.

- MD Sequential Compilation:

```
gcc -O3 -o MD_SEQ MD.c Particle.c Random.c ↩
    main.c -lm -Wall
```

- MD OpenMP Compilation:

```
gcc -O3 -fopenmp -o MD_OMP MD.c Particle.c↩
    Random.c main.c -lm -Wall
```

- MD MPI Compilation:

```
mpicc -O3 -o MD_MPI MD.c Particle.c Random.↩
    c main.c -lm -Wall
```

V. MD Paralelization study in shared memory environment with OpenMP

We've studied the original MD sequential algorithm, and we achieved a final modified algorithm that executes in $\mathcal{O}(n^2)$ too. In this work it's not actually supposed to implement a shared memory version of MD. However, we made it anyway to get an idea of MD algorithm scalability since we can make a quickly paralelization in shared memory thanks to OpenMP API.

After performing the analysis complexity and profiled all MD program we conclude that **force** is actually the most heaviest function consuming 99.89% of the total execution time. Therefore, we should explore the parallelism opportunities in force function because increasing the performance of this function by doing his computation in parallel it will globally allow us to achieve a bigger speedup. In the critical path represented in figure 2 we can see that **domove** and **mkekin** functions are also called 439400 times as **force** function. However, the paralelization of these two functions don't represent any speedup improvement – mainly due to their low computational power needs when compared to the whole kernel resources consumption. So the only function that actually will be paralelized will be **force** inside **cicleForces** function. The program calls a function for moving the particles and update their velocities, and after that **cicleForces** is called to compute forces. This computation we can made it in parallel. The for loops are a great source of paralelism, because we can divide the total iterations of the for loops for diferent threads and do the computation in parallel.

Above we can see the paralelization we made to **cicleForces** and **force function**:

basicstylebasicstyle

```
void cicleForces(MD *md,Particles *particulas){
  int i;
  md->epot = md->vir = 0.0;

  #pragma omp parallel for
  for (i = 0; i < md->mdsize; i++){
     force(md,particulas,i);
  }
}
```

With the *pragma* directive *omp parallel for* a team of threads is created and each thread executes **force** function in parallel. As we don't specify any scheduling policy, OpenMP performs a static scheduling. This policy is quite good if each *for* iterations loop takes about the same time to execute. Since different threads will execute the same function, we've to be careful if we need to update some memory values like array values or struct fields. That's whats happening in **third_newton_law** function when a thread updates a particle position with (x, y, z) coordinates and **update_static_variables** for updating **epot** and **vir** MD struct fields. Therefore this two functions calls must be performed in a critical area where only one thread can execute at a time:

basicstylebasicstyle

```
void force(MD *md,Particles *particulas, int pos){
    ...

 for (i = pos + 1; i < md ->mdsize; i++) {
    powDist[0] = calculateDistance(dist,posActPart,particulas,i,sideh,md);
    if(powDist[0] <= rcoffs)
    {
       calculePow(powDist);
       aux = powDist[7];

       leiNewton[0] = dist[0] * aux;
       force[0] *= leiNewton[0];

       leiNewton[1] = dist[1] * aux;
       force[1] *= leiNewton[1];

       leiNewton[2] = dist[2] * aux;
       force[2] *= leiNewton[2];
       #pragma omp critical
       {
        third_newton_law(particulas,i, leiNewton);
        update_static_variables(powDist, md);
       }
    }

  }
  #pragma omp critical
  {
   updateForce(particulas,force,pos);
  }
}
```

VI. MD Paralelization in distributed memory environment with MPI

I. MD implementation - Single Process Multiple Data

In this section we're going to explore all the necessary changes to the original code in order to produce an MPI implementation. We are going to produce an MD code where we split up the particles forces computation and run on multiple processors with different inputs in order to obtain results faster. That's a logic of *Single Program Multiple Data* (**SPMD**) technique.

In **main**, we change the original code with red changes:

basicstylebasicstyle

```
int main(int argc, char *argv[]){
  (.........)
  initialiseParticles(&md,&particulas);
  MPI_Init(0,0);
  GET_TIME(start);
    runiters(&md,&particulas);
  GET_TIME(end);
  elapsed = end - start;

 int rank;
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 if (rank == 0){
    printf( "%f\n", elapsed );
 }
  MPI_Finalize();
  JGFvalidate(md);
  return 0;
}
```

In the main we call **runiters** function as usual and measured the execution time. The MPI processes we create are going to comunicate through MPI_COMM_WORLD. The master process print the result with execution time.

The **runiters** will call **cicleForces** function:

basicstylebasicstyle

```
void runiters(MD *md, Particles *particulas){
  (.......)
  for (md->move = 0; md->move < md->movemx; ↩
       md->move++) {
    cicleDoMove (md,particulas);
    cicleForces (md,particulas);
    (.......)
  }
  MPI_Allreduce(MPI_IN_PLACE,&particulas->fX↩
       [0],md->mdsize,MPI_DOUBLE,MPI_SUM,↩
       MPI_COMM_WORLD);
  MPI_Allreduce(MPI_IN_PLACE,&particulas->fY↩
       [1],md->mdsize,MPI_DOUBLE,MPI_SUM,↩
       MPI_COMM_WORLD);
  MPI_Allreduce(MPI_IN_PLACE,&particulas->fZ↩
       [2],md->mdsize,MPI_DOUBLE,MPI_SUM,↩
       MPI_COMM_WORLD);
  MPI_Allreduce(MPI_IN_PLACE,&md->epot,1,↩
       MPI_DOUBLE,MPI_SUM,↩
       MPI_COMM_WORLD);
  MPI_Allreduce(MPI_IN_PLACE,&md->vir,1,↩
       MPI_DOUBLE,MPI_SUM,↩
       MPI_COMM_WORLD);
  MPI_Allreduce(MPI_IN_PLACE,&md->↩
       interactions,1,MPI_DOUBLE,MPI_SUM,↩
       MPI_COMM_WORLD);
}
```

After computing we performed a MPI **Allreduce** for (x, y, z) particles coordinates, ***vir*** and ***epot***. In the **cicleForces** function we split the **md->mdsize** iterations of force function calls through available processors.

basicstylebasicstyle

```
void cicleForces(MD *md,Particles *particulas){
  int i;
  md->epot = md->vir = 0.0;

  int rank, size;
  MPI_Comm_rank (MPI_COMM_WORLD, &rank)
      ;
  MPI_Comm_size (MPI_COMM_WORLD, &size);

  for (i=rank-1; i<md->mdsize; i+=size){
    force(md,particulas,i);
  }
}
```

So basically for a simple implementation of MD's **SPMD**, we only need to change the computation strategy on the functions wich corresponds to the critical path.

II. A theoretical analysis analysis to the All-reduce operation

In terms of operating results, an all-reduce operation is equivalent to a reduction operation that reduces the results to one process, followed by a broadcast operation that distributes the results to all processes, resulting in a Tree-based algorithm.

Specifically, let the P processes be denoted as $\text{proc}_0, proc_1, ..., \text{proc}_{P-1}$, t_s denote the communication latency, and N denote the total dataset size.

Therefore, after MPI wrap-up:

- For reductions: take P inputs and produce 1 output, resulting in $\mathcal{O}(t_s \times log\ P)$ time.
- For broadcast: take 1 input and produce P outputs, $\mathcal{O}(t_s \times log\ P)$ time.

Resulting in an All-reduce operation total time of $O(t_s \times log\ P)$.

For both communication techonologies Gigabit Ethernet and Myrinet 10Gbps, t_s will have a different value.

Notice that the linear model will also have to include communication start-up overheads. The following formula quantifies the commucation costs:

$$T_{Comm} = T_{Setup} + T_W(perbyte)$$

which derives into:

$$T_{Comm} = T_{Setup} + N \times t_s \times log\ P$$

II.1 An "runiters()" theorical model

Having described the Communication time model, we can go further in our theorical model and quantify the Computation time, resulting in the following formula:

$$T_{exec} = T_{Comp} + T_{Comm} + T_{Free}$$

In our case, the computation time will be the time spent on **force** by each process excluding communication synchronization time and free time. Every MPI process will calcute the force for its corresponding dataset subset, of size $\frac{N}{P}$, resulting in a computation time of:

$$T_{Comp} = \frac{N^2}{P}$$

The communication time, as stated earlier, is the time spent in send/receive data between processes.

The free time, is the time spent, when the processor becomes starved (not working). This time is difficult to model since it depends on the order of the tasks.

This time can be minimized, with adequate load distribution and/or adquate balance between computation and communication, resulting therefore in a "zero addition" to our model.

We can now derive the final theorical model wich results from the combining of the earlier derived formulas:

$$T_{exec} = \frac{N^2}{P} + T_{Setup} + N \times t_s \times log\ P$$

Where N will have the value 8788 for dataset size 1, and the value 19652 for the dataset size 3, and P will range from 1 to 128 MPI processes.

For the communication techonologies Gigabit Ethernet t_s will have the value of 32 μs, and for Myrinet 10Gbps t_s will have a value of 3 μs. We considered as T_{Setup} the value of 10 μs.

Given the prior formula we can infer a speedup estimation from:

$$Speedup = \frac{T_{exec}\ Sequential\ Version}{T_{exec}\ Parallel\ Version}$$

That can be derived into:

$$Speedup = \frac{N^2}{\frac{N^2}{P} + (\frac{N}{P} \times (T_{Setup} + t_s * log(P)))}$$

We can now infer a visusal representation of the potential speedup, as visible on figure II.1.

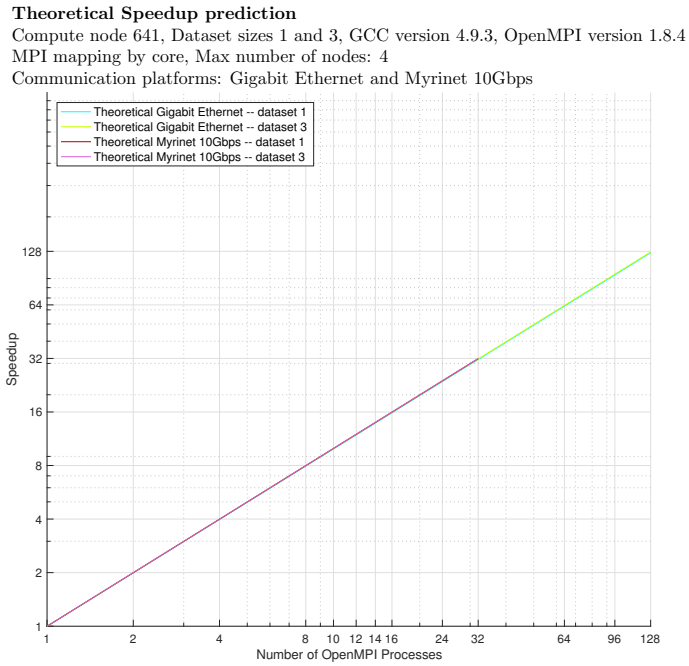

Figure 3: *Theoretical Speedup for the communication techonologies Gigabit Ethernet Myrinet 10Gbps for the datasets 1 and 3, for the parallel MPI version.*

VII. A graphical analysis on the MPI Algorithm implementation

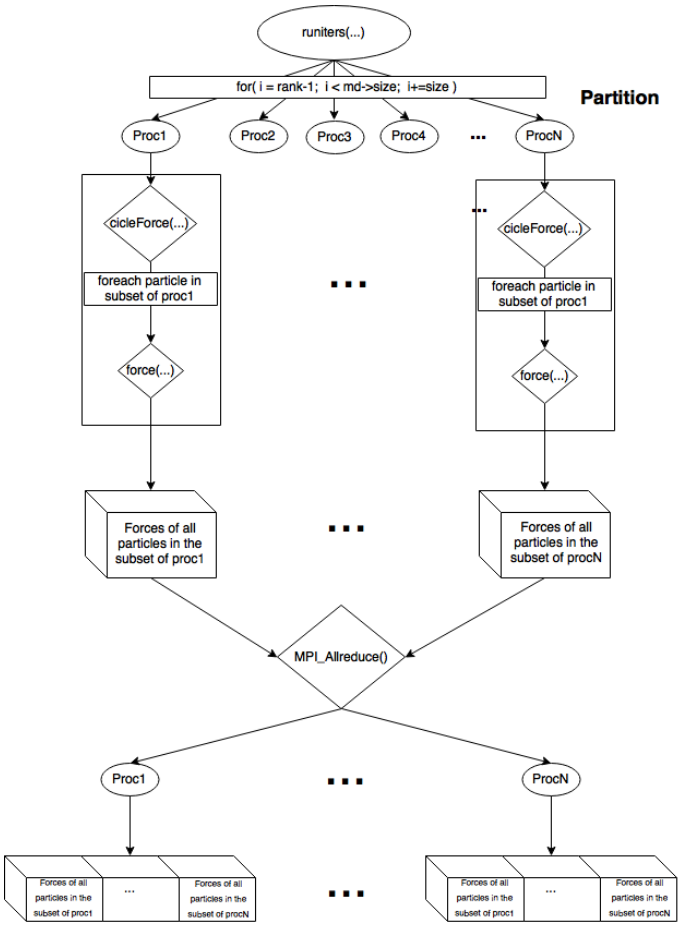

Figure 4: *Diagram of MPI Implementation*

In this diagram, we can analyse the behavior of the application with MPI implementation. As we can see, after start running the **runiters** function, it split the data-sets into subsets for differents processes, each processes run the **cicleForces** function, and foreach particle in subset it run the **force** function.

After the forces computation for all particles in the subset of each process, it's made a **MPI_Allreduce** for each process. In the final of execution, each process have all information of all processes.

VIII. SpeedUp Analysis with MPI implementation

After conclude the MPI MD implementation we made a scalability test. For testing our implementation we run the MPI algorithm using 4 nodes mapping by **core**. Using these 4 nodes on a **compute-641** we have a total 64 processes mapped in a physical cores. After 64 processes we're using hyper-threading allowing a total of **128 max processes**.

Figures 5 and 6 state the obtained results for time to complete the parallel region.

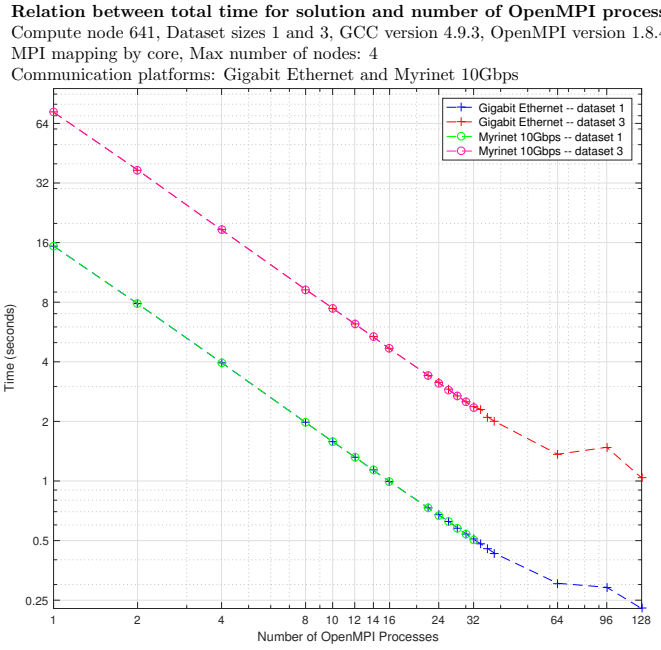

Figure 5: *Execution Time for ethernet and myrinet using 641 node*

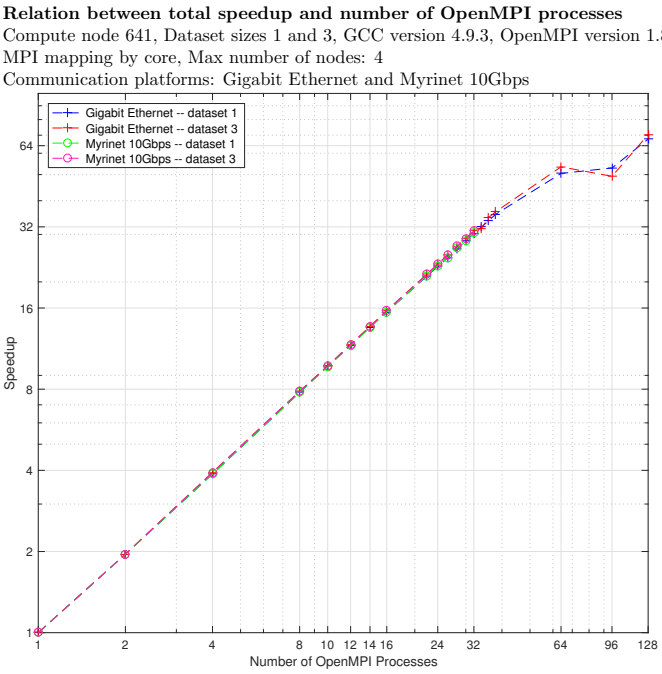

Figure 6: *Speedup for Myrinet and Ethernet using 641 node*

As we can state by figure 5 and 6 the time decreases and speedup increases almost linearly as we increase the number of MPI processes. We achieved a maximum speedup of 67.70× using 128 processes for dataset 1 wich corresponds to compute **8788 particles** using ethernet. For dataset 3 (**19652 particles**) we achieved a speedup of 70.25× using 96 MPI processes over ethernet connection. We tested two different comunication technology ethernet and myrinet. Globally we can say that the better speedups was achieved using ethernet. Even myrinet has much less latency and is faster than ethernet, doesn't allow us to get a better performance. That's because MD program and force algorithm performs much more computation than comunication.

We used **4 nodes** *compute-641* of SeARCH cluster so whereby up to 64 processes we've processes mapped in physical cores. When we use more processes than physical cores, we're using hyper-threading. The use of Hyper-Threading is beneficial because allow us to achieve the maximum speedup for dataset 1. However, for dataset 3 the *Hyper-Thread* causes a drop on the scalability when we use 128 processes instead of 64.

Given the obtained experimental results we should compare them with the theoretical prediction.

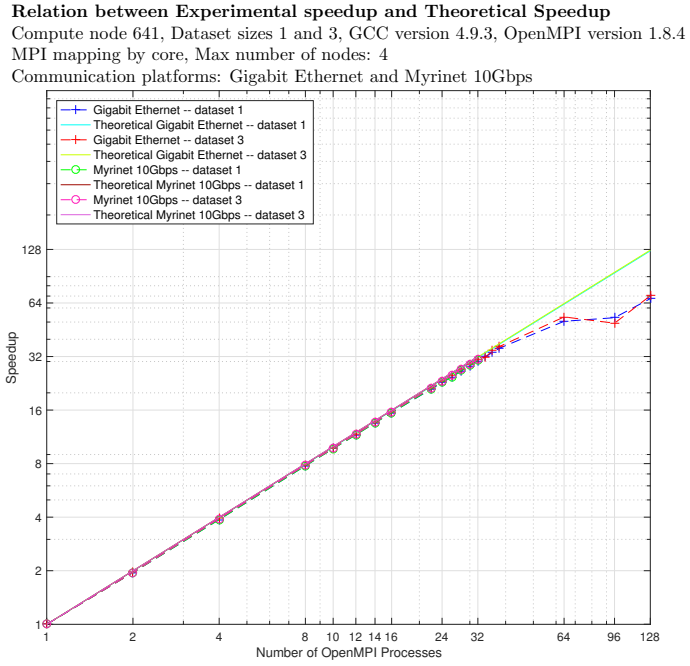

Figure 7: *Theoretical Speedup VS MPI implementation*

As we can infeer from figure 7, the experimental results approaches the theoretical predictions. The gap between potential speedup and obtained speedup for MPI processes ranging from 64 to 128 is easily explained due to the presence of hyperthreading. There are only 64 physical cores in the 4 nodes, resulting in the remaing 64 processes sharing CPU time with the other 64 processes.

IX. Conclusions

In this work, we performed a multi-core and multi-process MD implementation using **OpenMP** and **MPI** respectively. In the first phase of this work we've made a code analysis and a profile. We conclude that **force**, **domove** and **mkekin** are the three most called function in the algorithm with total of 439400 times each. From these three functions, **force** is the most

heaviest function with 99.89% of the total execution time. After studying the algorithm we concluded that the best sequential algorithm achieved executes in $\mathcal{O}(n^2)$.

We also made a modified version of the original sequential. We run the **perf** profiller to measure the impact of this modifications on the performance counters. The modified version show us a better results in some counters as cpu-cycles, cache-references, cache-misses and others. With MPI implementation we can conclude the algorithm scales linearly over ethernet and myrinet. With ethernet we could achieve a better performance since the algorithm perfoms much more computation than comunication. We achieved a maximum speedup of speedup of $67.70\times$ for computing **8788 particles** and $70.25\times$ for computing **19652 particles** in simulation.

[2] [3] [1] [4]

References

[1] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine. Open mpi: A high-performance, heterogeneous mpi. In *2006 IEEE International Conference on Cluster Computing*, pages 1–9, Sept 2006.

[2] Bruno Medeiros and João L. Sobral. Aomplib: An aspect library for large-scale multi-core parallel programming. In *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, ICPP '13, pages 270–279, Washington, DC, USA, 2013. IEEE Computer Society.

[3] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[4] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.