SHARED MEMORY PARALLELIZATION FOR MOLECULAR DYNAMICS SIMULATIONS OF NON-SPHERICAL GRANULAR MATERIALS

A. SCHINNER, K. KASSNER

Universität Magdeburg, Universitätsplatz 2, 39106 Magdeburg, Germany
E-mail: alexander.schinner@physik.uni-magdeburg.de
klaus.kassner@physik.uni-magdeburg.de

1 Introduction

The problem of the characterization of granular matter is not only a problem of material properties, but also a structural one. In this paper we use the molecular Dynamics to simulate interesting granular systems. Since simulations of this kind are time consuming, massively parallel computers such as the **CRAY T3E** are used. The domain to be simulated is divided into sub-domains, which are distributed to the different processors. Information about the boundary region is interchanged. In general, this is the bottleneck and slows down the program.

But there are alternatives. In our group multi-processor shared-memory machines, built by SUN, are available. Also multiprocessor Intel-based platforms using LINUX are becoming more popular. The objective of the work presented here was the implementation of a code which can be run on these cheap high-end shared memory workstations. The appropriate method for parallelization on this kind of machines is the use of *THREADS*.

For a more realistic simulation, we did not use spherical particles, but want to represent the particles as polygons.

So the program we present in this paper is a multi-threaded molecular dynamics simulation of 2-dimensional polygonal particles.

2 Programming with Threads

2.1 Shared Memory Architectures

Threads are a powerful tool for parallelization on appropriate computers, the are designed for shared-memory machines. We want to focus on symmetric multi-processor (SMP) architectures.[a] This architecture has some striking features. All processors are physically sharing the same memory and have a single address space. The processors do not communicate by sending messages across a network. They exchange, or rather *share* information, by writing to and reading from memory.

Looking at figure 1 one can see a schematic drawing of the hardware. The first problem to control is access to the memory, it is electrically nearly impossible[b] to allow simultaneous access to the same memory address. The next problem is the

[a]This kind of architecture is typical for "low-end"systems. Other systems have **NUMA** (Non-Uniform Memory Access), as used for example in the Cray T3E .
[b]Exceptions may be dual-port RAM's

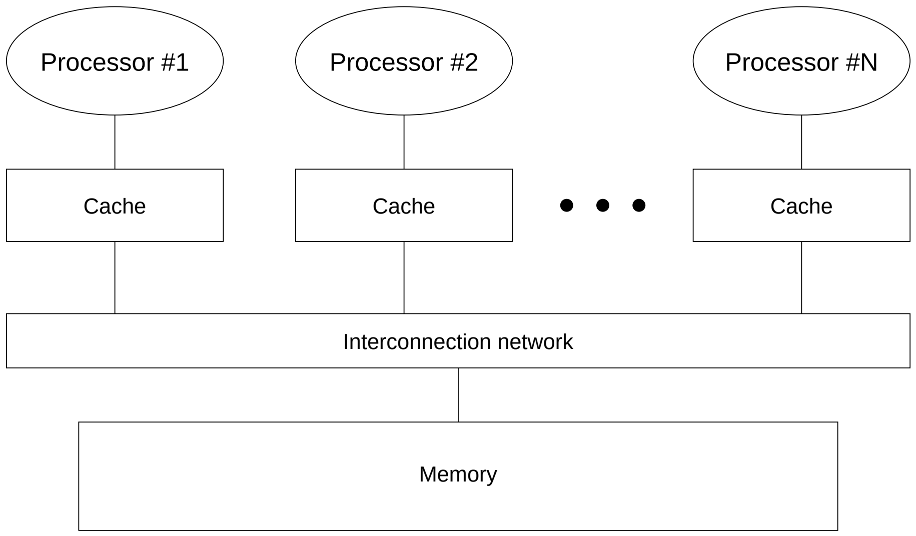

Figure 1. Schema of an SMP system

cache. Let us assume that a data is cached for processor # 1. Now processor #2 changes these data in the memory. Then the data in the cache of processors #1 has to be declared invalid. Although this is not the programmers problem, it is good to know these basic problems, since similar problems may arise while using threads on the programming side.

2.2 Threads

A thread of control, or more simply a *thread*, is an independent sequence of execution of program code inside a UNIX process. All threads share the memory of the same process. The threads within a process are scheduled and executed independently in same same way as normal UNIX processes. On multiprocessors, different threads may be executed on different processors.[1,2]

A program is a sequence of commands. To execute a program, we have a small unit executing these string of commands in a given order. But why only one executing unit? Imagine two units, working simultaneously on different parts of the same program. Then these units are called threads. A program starts with the main process. Then the programmer can create threads, which start to execute a given function with given data. We want to explain this using the following example.

```
#include <errno.h>
#include <pthread.h>
#include <stdlib.h>
#define LENGTH 1024

/* Set value of every element  */
void *worker(void *input){
    int *ptr;
    int i;
    ptr=(int *)input;  /* cast to appropriate type */
    for (i=0;i<LENGTH;i++) ptr[i]=i;
    return(NULL);
}

int main(int argc, char **argv){
    /* Declare variables */
    int data1[LENGTH],data2[LENGTH];
    pthread_t thread;
    pthread_attr_t thread1_attr;

    /* necessary initializations */
    pthread_attr_init(&thread1_attr);
    pthread_attr_setscope(&thread1_attr,PTHREAD_SCOPE_SYSTEM);

    /* serial execution */
    worker(data1);
    worker(data2);

    /* parallel execution */
    pthread_create(&thread,&thread1_attr,worker,data1);
    worker(data2);
    pthread_join(thread,NULL);

    exit(1);
}
```

The function `worker` does the time consuming work by manipulating a vector of integers. So, if this work has to be done for a different vector, we call `worker` as often as necessary. In a serial program, this is sequentially. But obviously this work can be done in parallel. By calling `pthread_create(&thread, &thread1_attr, worker, data1)` we ask the operating system to generate a thread which starts to work with the function `worker` and the parameter `data1`. Immediately after creating this thread, we can continue by calling `worker` directly with parameter `data2`. Remember, while we do this, the other thread is still working with `data1`. At the end we call `pthread_join(thread,NULL)` to wait for the thread to be completed. (Fig. 3)

Inside of a thread we can of course call other functions, allocate memory and so on like in a normal C-program.

If the data we want to manipulate is independent for each thread this is all one has to know about threads. However the simulation is more complicated if different threads access the same data. For instance:

```
for (i=0;i<LENGTH;i++){
   z=z+f(i);
}
```

Assume that we split the loop into two parts and each part is calculated by one thread. Then the following sequence is possible:

step	thread #1	thread #2
1	copy the value of z to cpu #1	
2	calculate f	copy the value of z to cpu #2
3	add local copy of z and f	calculate f
4	copy local copy of z to memory	add local copy of z and f
5		copy local copy of z to memory

The value calculated by thread #1 is lost! So this code will give unpredictable errors, we have "race conditions".

We have to protect shared resources, if we want to modify them. We need to use a synchronization called *mutual exclusion* or *mutex* for short. If one thread has exclusive access to data, no other thread can simultaneously access the same data. The *mutex* variable is also be called a *semaphore*. In the former example one can declare a *mutex* variable `pthread_mutex_t mutex_for_y;`, which controls the access to variable z.

Then the body of the loop would look like this:

```
temp=f(i);
pthread_mutex_lock(&mutex_for_y);
y=y+temp;
pthread_mutex_unlock(&mutex_for_y);
```

Then the execution sequence could look like this:

step	thread #1	thread #2
1	calculate f	
2	lock mutex	calculate f
3	get local copy of z	wait for mutex
4	add temp and local copy of z	wait for mutex
5	copy local copy of z to memory	wait for mutex
6	unlock mutex	wait for mutex
7		lock mutex
8		get local copy of z
9		add temp and local copy of z
10		copy local copy of z to memory
11		unlock mutex

Even though there are much more steps involved if the calculation of f is really time consuming we obtain a good speedup.

In the case that both threads only read from the same variable and don't want to modify it, no semaphore is needed. This is only necessary if at least one thread may change the variable's value.

`pthread_create`, `pthread_join` `pthread_mutex_lock` and `pthread_mutex_unlock` are all we need for writing a full featured parallel program.

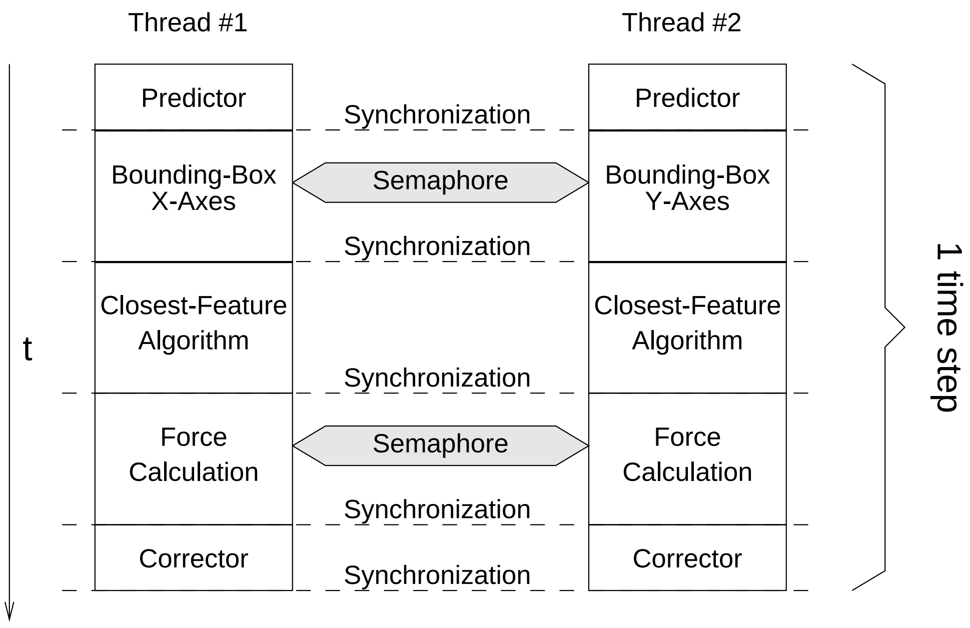

Figure 2. Basic structure of the algorithm for two threads.

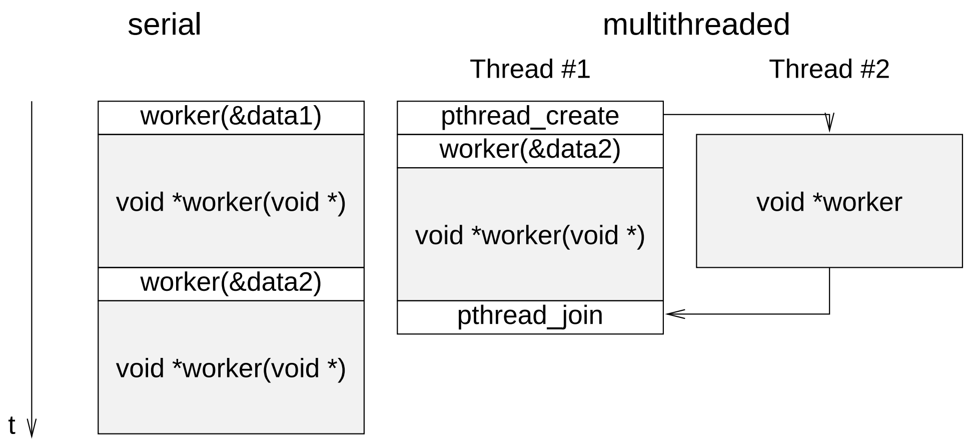

Figure 3. Parallel and serial version of the example.

3 Threads and Granular Matter

3.1 Basic Strategies

The most time-consuming part of a molecular dynamics simulation is the computation of the forces. As we calculate the force between two particles from the overlap, we have to determine all collisions in our simulation.

To be able to efficiently simulate particles on a large scale we use a bounding box-algorithm, which updates the list of colliding bounding-boxes in $\mathcal{O}(n)$. This will dramatically reduce the number of possible collisions. In the next step we calculate for each possible collision the distance between the surfaces of the particles involved. Since we have an algorithm which can calculate this value within constant time (independent of the complexity of the particles) we can generate the list of collisions very fast

We have to calculate the area of overlap only if the overlap area is larger than zero. This saves a lot of time, since calculating the overlap is time consuming, especially if the overlap area is zero.

The most important thing for efficient programs is the organization of the data structures. We have three fundamental structures. One structure is for the representation of the particles. Here we have variables like position, velocity and shape. These structures are organized in a linked list. The second structure is for the bounding boxes. Here we have two linked lists, one for the x-axis, one for the y-axis. The third type of structure represents a collision. This structure is for both bounding box collision and particle collision. If we find an collision for particle i and j $(i < j)$ we keep the structure for this collision in a small linked list, appended to the structure for particle i. So we can search for a collision much faster than in a long global list.

The structure for the particles has a `mutex`. This enables us to protect either the particle's information itself or the appended list of collisions.

All algorithms are optimized for using information from the last time step. The reason is that for molecular dynamic simulations the particles move a rather small distance between two steps. Hence the information gained in the last step is not very wrong and can be used as a good approximation for the current time step. More information on the algorithms will be available[3], Matuttis[4] described the force calculation in detail.

3.2 Bounding-Box Algorithm

The first step is the bounding-box algorithm. This part tries to reduce the number of possible pairs of particles which have to be checked for collision.

We use an algorithm which is basically sorting the bounding box boundaries along one coordinate axis. We have rules telling us how to manipulate the list of collisions, depending on steps necessary for resorting. For the sake of simplicity, we sub-divided the code into two threads, one for each axis. Since the information on each collision can be changed by both threads, we have to use the particles semaphores to guarantee the integrity of the data. However, most of the time, the two threads do not have to wait. The result is an actual list of *possible particle-particle collisions.*

3.3 Closest-Feature Algorithm

The closest-feature algorithm is a fast method for determining the distance of two convex polygons. We take advantage of the fact that calculation time is generally independent of the number of vertices of the polygons.

Since the lists of possible particle-particle collisions are appended to the list of particles, we divide the list of particles into sublists for independent processing. To calculate the distance, the threads have to read only from the structures of both particles, which are not changed in this step. The information is written to the structure of the collision, but it is guaranteed that no other thread reads information from there. So we do not need any semaphore, the both threads are completely independent.

When both threads are finished, we have the information about which particles are colliding, stored in the structures of the collisions.

3.4 Force calculation

In the next step we have to determine the forces from each collision. Therefore we calculate the overlap area for each particle pair, and will do further calculations for which we need information stored in the structure for the collision.

Here again we split the list of all particles and each thread works on the according collisions. All information needed for the calculation of the force is not changed in this step, reading information does not require any `mutex`. But we have to sum up the total force for each particle. Since it is possible that another thread is also working on a collision the particle is involved in we have to protect the force with the particle's `mutex`. However, since this is unlikely we do not lose much time having waiting threads.

3.5 Predictor-Corrector Differential Equation Solver

We use a Gear predictor-corrector method for solving the differential equation[5]. The first step predicts position and velocity for the particles. Then one has to calculate the forces and can correct position and velocity. Both predictor and corrector only read and write information inside of a particle's structure. So we do not need to protect the data by semaphores. Hence we again split the list of particles and let them be calculated by threads.

4 Simulations and results

Building a sandpile by pouring material from a point source is a very interesting experiment because under certain conditions that below the apex of the sandpile the ground pressure has a pronounced dip. This is a fascinating, although heavily debated, behavior [6,7,8].

To explain this effect different continuum theories have been developed. However, these theories introduce additional assumptions for for the stress tensor. Since it is nearly impossible to measure the stress inside a sandpile we have performed simulations with our parallel code. We want to present one of these simulation runs here.

We poured a mixture of small and big roundish particles onto a floor. Overall there were 4500 particles with 7 corners each. Figure 4 shows a picture of the pile. The different shading represent the time when the particles were dropped onto the pile. The asymmetries are due to avalanches during the building process. Calculating the pressure on the ground we find a dip (Fig. 8). Whereas these results may be compared with experiments, we can also calculate quantities which are not available to the experimentalist.

In figure 4 we see the force network inside the pile. On the bottom of the pile we can see the endings of two main force paths left and right to the middle. Having the forces it is simple to calculate the stresses inside the pile (Fig. 6). Determining the angle of the main axes for different layers of the pile, we see that the main axes are not parallel but are changing smoothly from the left to the right side. Other interesting information such as isobars (lines of equal σ_{zz}) may be calculated, too (Fig 9,10). Movies showing the simulation are available at `http://itp.nat.uni-magdeburg.de/~schinner/granular/movies.shtml`.

5 Conclusion

The speedup for programs on two processors is about 1.9 to 1.95 depending on the numbers of particles. These values have been measured under working conditions, the computer doing this simulation was also working as file server.

We have presented our parallel versions of algorithms for the simulation of granular materials. There is no need to interchange any information between the threads. The program has been designed for offering optimal conditions for shared memory parallelization.

We would like to thank Hans-Georg Matuttis for fruitful discussion. We are indebted to the organizers of this workshop who provided us with the possibility for a more elaborate description of our work.

References

1. S. Kleinman, D. Shah, and B. Smaalders. *Programming with Threads.* SunSoft Press A Prentice Hall Title, 1996.
2. B. Nichols, D. Buttla, and J. P. Farrell. *Pthreads Programming.* O'Reilly & Associates, Inc., 1997.
3. Alexander Schinner. Fast algorithms for the simulation of polygonal particles. submitted to *Granular Matter.*
4. H.-G. Matuttis. Simulations of the pressure distribution under a two dimensional sand-pile of polygonal particles. *Granular Matter*, 1(2):83–91, 1998.
5. M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids.* Oxford University Press, 1987
6. Tomosada Jotaki and Ryuichi Moriyama. On the bottom pressure distribution of the bulk materials piled with the angle of repose. *J. of the Soc. of Powder Technol., Japan*, 16(4):184–191, 1979. in japanese.
7. J. Šmid and J. Novosad. Pressure distribution under heaped bulk solids. *I. Chem. E. Symposium Series*, 63:D3/V/1–12, 1981.
8. J. Šmid. Druckverteilung unter einem Schüttguthaufen. *Grundlagen der Landtechnik*, 33(3):72–75, 1983.

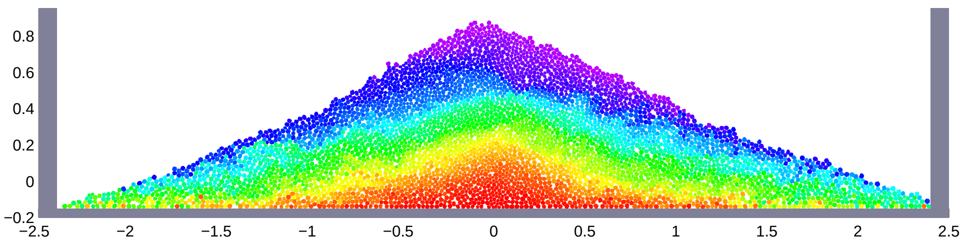

Figure 4. Picture of the pile, build from the point source. The shading represent the time the particles have been dropped in.

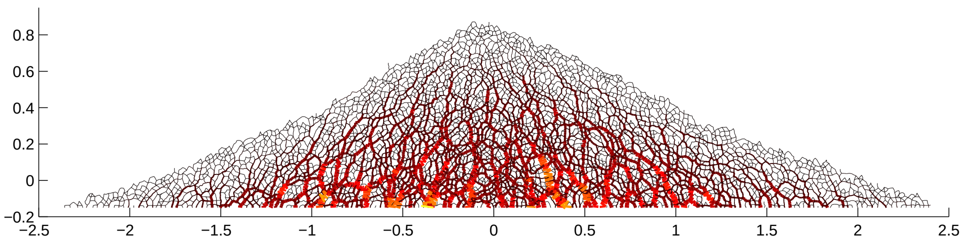

Figure 5. Picture of the force-net inside of the pile. Stronger forces are brighter and have wider lines.

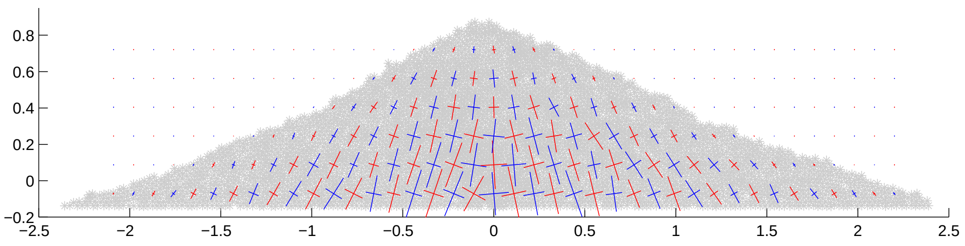

Figure 6. Main axis of the stress tensor inside the pile.

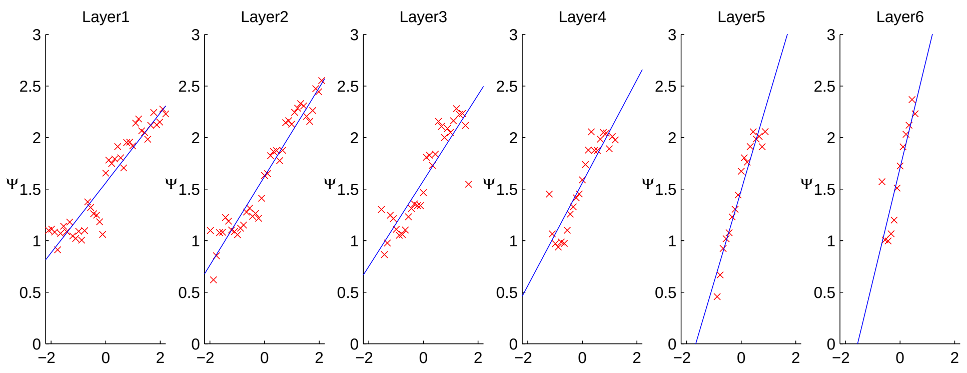

Figure 7. Angle of the stress tensors' main axis for different layers.

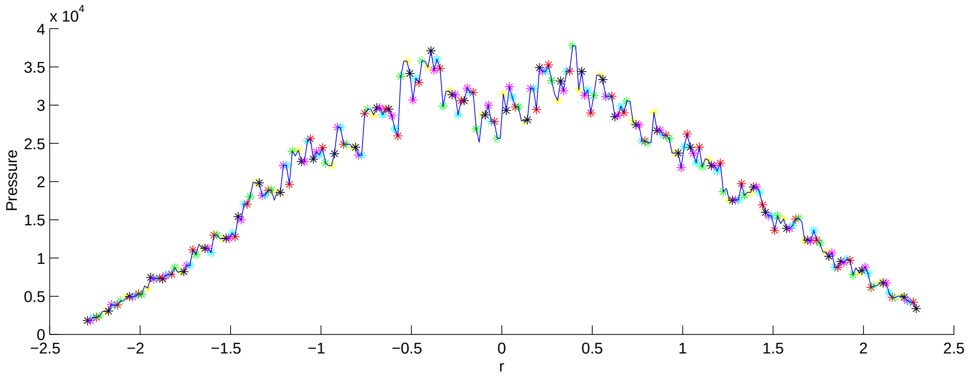

Figure 8. Pressure onto the ground. One can see the Minimum of the pressure beneath the center of the pile.

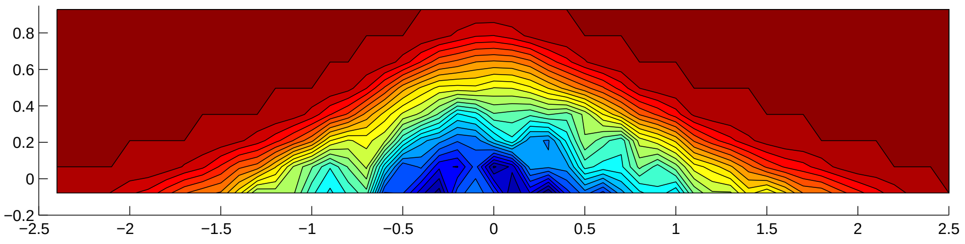

Figure 9. Isobaric lines inside the pile. Red areas have low pressure, blue areas are areas of high pressure ($\approx 10^6 N/m$).

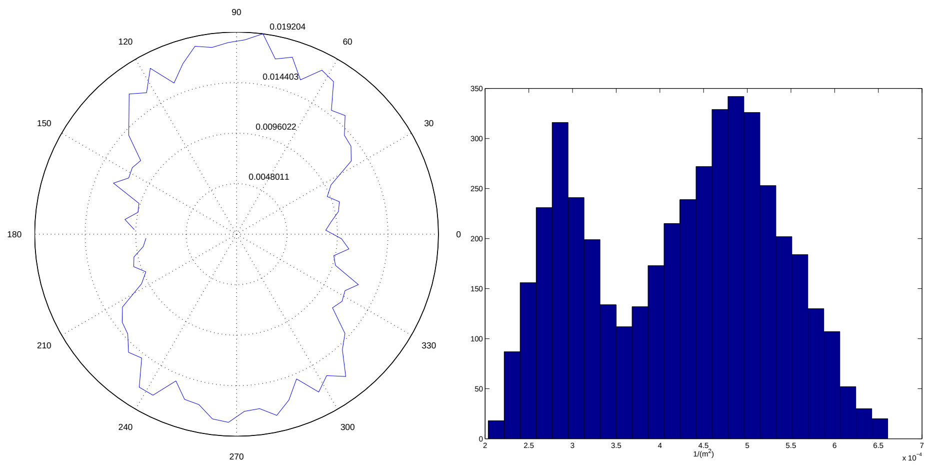

Figure 10. The left side shows a polar plot distribution of the angles of all forces. The right side is the histogram for particles area, one can see, that we used a bidispers mixture.