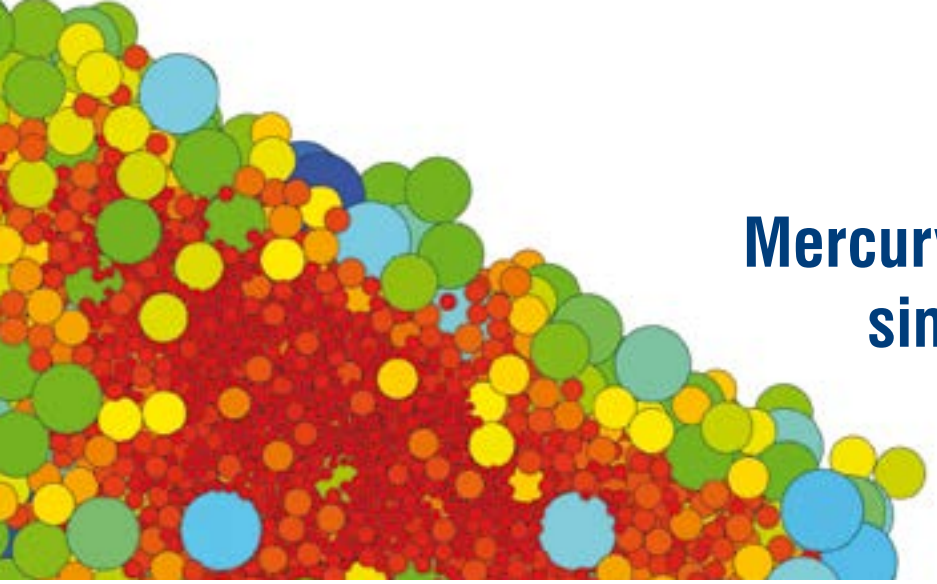

Mercury-DPM: Fast particle simulations in complex geometries

Mercury-DPM is a code for performing discrete particle simulations. That is to say, it simulates the motion of particles, or atoms, by applying forces and torques that stem either from external body forces, (e.g. gravity, magnetic fields, etc…) or from particle interactions. For granular particles, these are typically contact forces (elastic, viscous, frictional, plastic, cohesive), while for molecular simulations, forces typically stem from interaction potentials (e.g. Lennard-Jones). Often the method used in these packages is referred to as the discrete element method (DEM), which was originally designed for geotechnical applications. However, as Mercury-DPM is designed for simulating particles with emphasis on contact models, optimized contact detection for highly different particle sizes, and in-code coarse-graining (in contrast to post-processing), we prefer the more general name discrete particle simulation. The code was originally developed for granular chute flows, and has since been extended to many other granular applications, including the geophysical modeling of cinder cone creation. Despite its granular heritage it is designed in a flexible way so it can be adapted to include other features such as long-range interactions and non-spherical particles, etc.

Why a new simulation code?

There are many open-source particle simulation packages, so the question arises of why another? Mercury-DPM was originally started as a joint collaboration between the Multi-Scale Mechanics (MSM) and the Mathematics of Computational Science (MaCS) groups (before 2011 they were called the Numerical Analysis and Computational Mechanics) at the University of Twente in 2009. The idea was to develop a code that could be used alongside the existing MaCS group continuum solver hpGEM (http://wwwhome.math.utwente.nl/~hpgemdev/) to approach problems using various multi-scale computational methods. Around the same time Vitaly Ogarko and Stefan Luding developed an advanced contact detection method: the hierarchical grid. This novel algorithm is quicker than existing methods for poly-dispersed flows (and still the same speed for mono-dispersed). So the idea of a new simulation code that had three core design aims was born:

1. It should be easy to use with minimal C++ knowledge.
2. It should be built around the new hierarchical grid detection method.
3. It should be able to generate accurate continuum fields that could be used with/alongside continuum solvers.

Actually, the name of the code emanates from the contact detection method: hierarchical grid → Hgrid → Hg → Mercury.

Features

Since it was first started it has evolved and gained many novel features. The main features include:

1. Of course, the hierarchical grid: The neighborhood search algorithm to effectively compute interaction forces, even for highly poly-dispersed particles.
2. Built-in coarse-graining statistical package: it has an in-built advanced statistics package to extract continuum fields such as density, velocity, structure and stress tensors, either during the computation or as a post-processing step.
3. Access to continuum fields in real time: The code can be run in live statistics mode, which means it can respond to its current macroscopic state. An illustrative example of using this would be a pressure-release wall, i.e., a wall whose motion is determined by the macroscopic pressure created by particle collisions and moves such that its pressure (not position) is controlled.
4. Contact laws for granular materials: many granular contact force models are implemented, including elastic (linear or Hertzian), plastic, cohesive, sintering (temperature/pressure/time-dependent), and frictional (sliding/rolling/torsion) forces.
5. Simple C++ implementation: Mercury-DPM consists of a series of C++ classes that are flexible, but easy to use. This allows the user to generate advanced applications with only a few lines of code.
6. Handlers: The code has handlers for particles, walls and boundaries. Thus, each object type has a common interface, even though individual objects can have completely different properties. This also makes it easier for the user to create new objects.
7. Complex walls: The code not only supports simple flat walls, but also axial-symmetric, polyhedral and helical screw walls are available. Additionally, due to the handler interface it is easy for more advanced users to define new types of walls themselves.
8. Specialized classes: Many specialized classes exist that reduce the amount of code required by the user to develop standard geometries and applications. Examples include chute flows, vertically vibrated walls and rotating drums.
9. Species: Particles and walls each have a unique species, which

is hidden for basic use of the code; however, this feature can be enabled by a single function call. Different particle properties for each species and different interaction forces for each pair of species can then be defined, allowing the simulation of mixtures.

10. Self-test suite and demos: Mercury-DPM comes with a large (over 100) self-tests and demo codes. These serve two purposes: 1) they allow us to constantly test both new and old features so we can keep bugs to a minimum; 2) they serve as tutorials, for new users, of how to do different tasks.
11. Simple restarting: every time a code is run (and at intervals during the computation) restart files are generated. Codes can be restarted without recompilation simply by calling the executable again with the restart file name as an argument. Also the restart files are complete in the sense that they contain all the information about the problem. In this way, small changes can be made (e.g. with the individual particle density or the coefficient of restitution) and the simulation can be rerun without the need for recompilation of the code.
12. Visualization: The particles output can be visualized easily using the free package VMD (visual molecular dynamics, http://www.ks.uiuc.edu/Research/vmd/).
13. Parallel: Currently a parallel-distributed version of the code is under development using MPI and this version should be publicly available shortly.

Simple C++ implementation and handlers

Mercury-DPM is a very versatile, object-oriented C++ code, which means new applications can be developed rapidly and easily. It has been tested for several Linux distributions and Mac OS. It consists of a core (kernel) that contains a series of C++ classes onto which users can quickly build to develop their own application (driver). The base class, Mercury3D, is flexible and contains the basic functionality to define a simulation. Using this class, the users specify the particulars of their simulations (initial positions, inflow, outflow, walls, interaction parameters, etc.) in a single driver file, which calls the kernel to perform the simulation. In addition to the flexible base class many higher-level, more powerful classes exist, which are tailored for common problems. A typical example would be the class Chute. This automatically defines a bottom that can be smooth or rough (of which we have three different types), an inflow boundary, outflow conditions, sidewalls, etc. and gives the user new access functions to perform standard tasks; for example: set_ChuteAngle (which automatics rotates the gravity vector), and set_InflowHeight (which changes the height of the particle layer at the entry to the chute). These common functions allow the simple setup of chute flow problems in just a few lines of code. Many other high-level classes exist and a full up-to-date list can be obtained from the website, http://www2.msm.ctw.utwente.nl/MercuryDPM. As the code is fully object-oriented, many of the classes build on each other adding extra levels of functionality. An example would be the class ChuteWithHopper, which replaces the inflow conditions in the original Chute class with a more complicated hopper construction. In addition, it adds new access functions, which allow the hopper properties to be set. Due to the object-oriented nature of the code it is easy for users to change a driver code from one class to a similar one. For example to change a Chute problem to a ChuteWithHopper problem all the user has to do is change the class he includes at the top of the code and replace the access functions like set_InflowHeight to the hopper equivalent i.e. set_HopperWidth, set_HopperHeight, etc. All the code defining the geometry and dealing with particle properties does not have to be changed and has exactly the same interface.

Another key feature of the Mercury-DPM design is the idea of handlers. There are three handlers in Mercury-DPM: Particle, Wall and Boundary. Handlers mean that all items of the same basic type are stored in one place. This has several nice advantages the primary being the flexibility, i.e., each particle, wall, etc., can have completely different properties but as long as the basic properties are defined the code can deal with the item. The user does not have to look after the walls, particles and boundaries themselves; they only have to create them. For example, to add a new particle to the simulation the user defines the properties of the new particle and passes them on to the ParticleHandler, then the code does the rest. The user does not need to know anything about other particles that have previously been created. The handler can also be queried via its access function to obtain information like the number of particles currently in the simulation, the smallest particle, etc.

Applications

Here we will illustrate some of the features of Mercury-DPM via applications that have already been developed in the package.

Poly-dispersed segregation in a rotating drum (S. Gonzalez, S. Luding, A.R. Thornton)

One of the most fascinating properties of granular matter is the ability of appropriately driven mixtures to separate into their individual components, despite the apparent lack of energetic or entropic advantages of a segregated state. This often produces brilliant patterns that give rise to a number of interesting problems in nature and difficult challenges for the powder compressing. The segregation of a binary mixture contained in a partially filled, horizontal, rotating drum is an extensively studied problem of this class; one with obvious industrial importance. One of its most beautiful characteristics occurs when it segregates in the radial direction, producing a core rich in small particles surrounded by an outer layer of mainly big particles, and depending on the angular velocity, rich patterns of segregation. Despite the great number of studies involving two-components systems, poly-disperse systems remain mainly unexplored, although they are more the rule than the exception in nature. The importance of these systems for industry is obvious; from a theoretical and computational point of view, they present various and difficult challenges.

One of the key reasons why poly-dispersed flow has not been investigated in the past is the computational cost. Traditionally, particle simulation codes use a linked list system for contact detection. This method has a complexity of order N for mono-dispersed flows, where N is the total number of particles. This means that if you double the number of particles the total computational time doubles. However, for poly-dispersed flows this nice scaling is lost and in the extreme limit of one very large particle and the rest containing small particles, the complexity

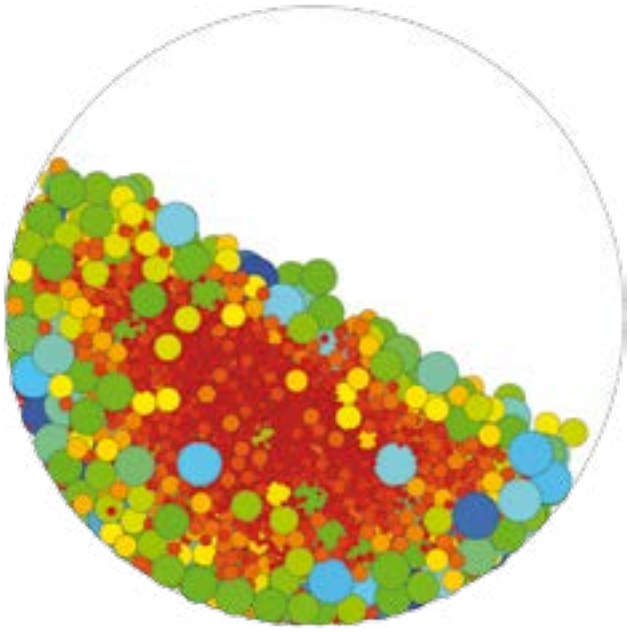

Fig. 1 - Poly-dispersed segregation in a rotating drum. Colour denotes particle size

becomes order N2. Yet due the hierarchical grid contact detection algorithm that forms Mercury-DPM's heart this problem is still order N within Mercury-DPM. This is why highly poly-dispersed and wide-size distributions can be easily tackled for the first time, in an open source environment.

Our simulations consist of spherical particles with different size distributions. Fixing particles to the surface of a given geometry makes the walls of the rotating tumblers. An easier way is to define finite walls; this is done in the driver. Turning the angle of gravity simulates the rotation of the tumbler. This makes the simulation with finite walls easier, and since the speed of rotation is quite low, the approximation is valid. For higher speeds, centrifugal forces have to be considered or the walls moved (which is possible within Mercury-DPM). Figure 1 shows an example of one of these poly-dispersed simulations. In this simulation every particle is of a different size, with a uniform volume distribution. The color represents the size of the particles with red the smallest and blue the largest. The ratio of the smallest to largest particle is ten to one in this simulation. The image is taken after two revolutions of the drum, and a strong segregation pattern can be observed with the small particles located in the center of the drum.

Granular flow through a contraction (D. Tunuguntla, A.R. Thornton, T. Weinhart, O. Bokhove)

As a stepping-stone towards analyzing complex granular flows in industry, we analyzed flow in an inclined channel with a contraction. In order to simulate steady-state flow through the contraction, the flow should be in steady state when entering the contraction. Regular inflow conditions such as the insertion of particles at the boundary or through a hopper would require us to simulate a large stretch of flow before the contraction to obtain steady flow at the beginning of the contraction. In order to reduce the

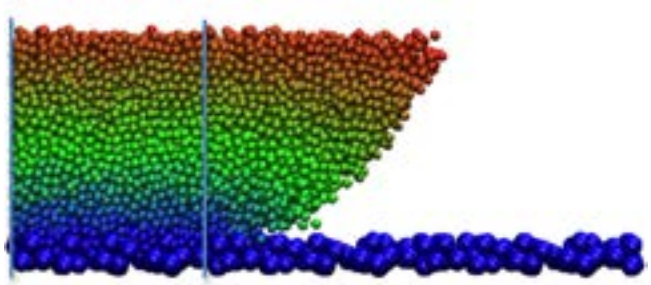

Fig. 2 - Illustration of the Mercury inflow maser. Colours indicate particle speed, with blue low and red high speed. Lines indicate the modified periodic boundaries.

computational costs to a minimum, a new special type of inflow has been designed that produces steady uniform flow directly at the contraction entrance. This is done using a small periodic box in the inflow regions, the downstream wall of which both mirrors and transmits the particles into the main chute. That is, each time a particle moves through the downstream periodic wall, a copy is created which ignores the periodic walls and thus flows into the contraction. This inflow type was named maser, as it acts as a material laser, creating a steady uniform inflow of particles. Meaning that a small cheap steady-state periodic-box simulation can be used to seed the much larger simulation through the contraction. Details of this kind of inflow will be presented in a later publication. An illustration of this inflow is shown in Figure 2, here the flow is visualized in VMD (visual molecular dynamics, http://www.ks.uiuc.edu/Research/vmd/); Mercury-DPM contains wrappers to view its output in this package.

Mercury-DPM contains an implementation of arbitrary convex polyhedral walls. These walls have been carefully designed to ensure that the collision with each face, edge, or corner of the wall is treated correctly. The main difficulty here is to determine the nearest face, edge or corner, and the normal direction of each collision. In a particle-face collision, the normal always equals the face normal; whereas, the normal of an edge-particle collision depends on the position of the particle with respect to the edge; finally, the collision with a corner is equivalent to colliding with an infinite mass particle.
Figure 3 shows a simulation of the granular flow through a contraction formed by two polyhedral walls. Once the particles flowing down the channel enter the contraction, jumps/shocks

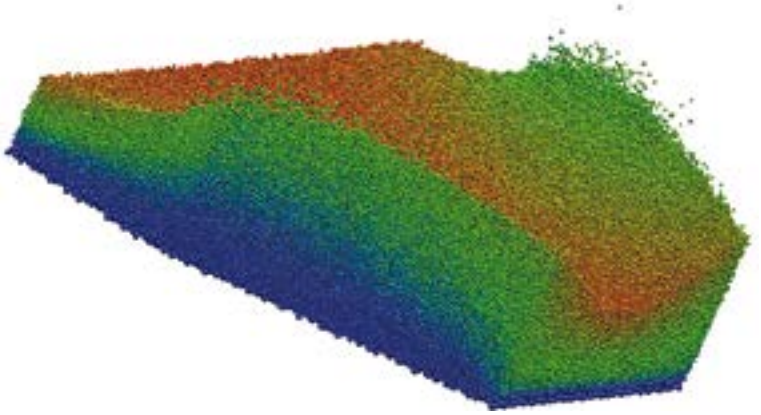

Fig. 3 - 380,420 particles flowing through a contraction. Colours indicate particle speed.

in depth and velocity profiles are observed. The interaction of the particles with the sidewalls of the contracting channel can be seen in Figure 3, where the colour denotes the speed. The blue region illustrates the jump in the velocity profile of the flow.

Granular jet impacting of an inclined plane (R.H.A. Fransen, A.R. Thornton, S. Luding, T. Weinhart)

Here, we simulate a granular jet impacting an inclined plane using Mercury-DPM. This problem was first investigated both experimentally and via the continuum approach by Johnson and Gray.

The novelties in the implementation are the construction of inflow conditions through a funnel and the modeling of a rough surface. Finally, the depth-averaged height and velocity are extracted from the simulation using our coarse-graining toolbox.

To obtain a jet of particles, a funnel is created using fixed particles placed onto a conical shape. Particles are inserted into the top third of the funnel whenever a free space is detected, see Figure 4 top left. The roughness of the funnel wall is necessary to create a velocity profile that keeps the developing jet from spreading.

To obtain the strong frictional effects observed in experiments, the plane needs to be rough as well. Therefore, a disordered layer of fixed particles is created. To prevent particles from falling through, a planar wall is placed below the fixed particles. Using the frictional rough walls allows us to observe similar profiles of the impact and fast-flowing zones as in the experiments, see bottom left and right in Figure 4.

Particles are removed from the simulation when they reach the end of the plate. This leads to a low pressure at the outflow, which can affect the flow on the plate. In Mercury-DPM the user can define a removal condition. If this is set to be a few cm below the plate, that is, when the particles are in a free flowing jet, off the end of the plate, it has been shown not to have an affect on the main flow.

One of the major novel features of Mercury-DPM is its coarse-graining toolbox, which constructs a continuous macroscopic field from the discrete particle data. This toolbox can be both run as either a post-processing step or live during the simulation. Careful attention has been paid to the boundary areas, and this package is even able to produce continuum fields within one particle diameter of a boundary. Examples of the results of the course-graining package for the jet problem are shown in Figure 5.

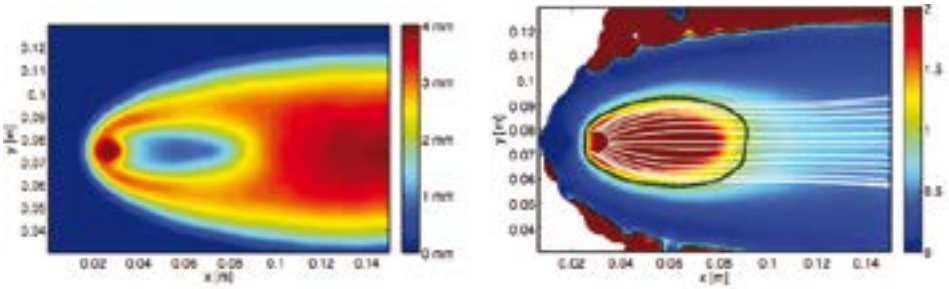

Fig. 5 - Course grained macroscopic fields created using Mercury-DPM's coarse-graining toolbox. Left shows the height of the flow in millimetres and right the local Froude number of the flow. The white lines indicate velocity streamlines; the black line indicates the location of a hydraulic jump/shock.

In order to obtain the height of the flow we assume that the density of the flow is constant over height, and that the flow is steady and uniform enough to have a lithostatic stress profile, see Figure 5. Thus, the height can be defined using the depth-averaged stress and density as plotted in Figure 5. Once the height is known, a depth-averaged velocity and the Froude number can be defined. A Froude number larger than unity denotes supercritical flow, otherwise the flow is subcritical. This allows us to determine the location of the shock (black line in right panel of Figure 3).

Screw feeder and conveyor (D. Krijgsman)

The final feature of Mercury-DPM we will illustrate in detail is the helical screw. This highlights the flexibility of the versatile handlers, they are common in many industrial apparatuses. The difficulty of these simulations lies in the interaction between the screw and the particles. The approach that is used in most similar particle simulation packages is to triangulate the screw and do collision detection between the particles and small segments of the screw. The major disadvantage of this method is that for accuracy the single screw element has to be divided into a large number of triangles. All possible combinations of these triangles with the particles have to be checked for contacts, resulting in high computational costs. To circumvent this, in Mercury-DPM the screw is modeled as a single parametric surface.

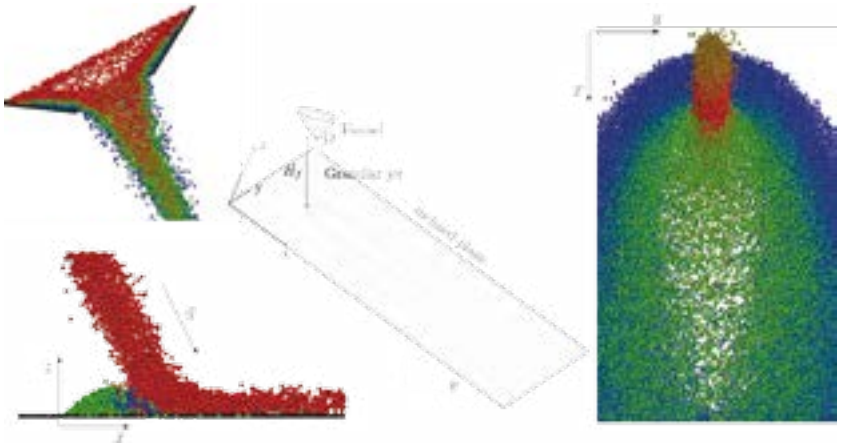

Fig. 4 - Top left shows the flow in the hopper, bottom left the impact region, middle schematic of the original experiment, right the top view of the full particle simulation (~500k particles). Black particles indicate fixed particles; all other colours indicate speed, with blue low and red high speed.

In Mercury-DPM the screw is treated as 'just another wall' so all the user has to do is to create a screw and pass this screw to the WallHandler. The code automatically deals with the collision detection and

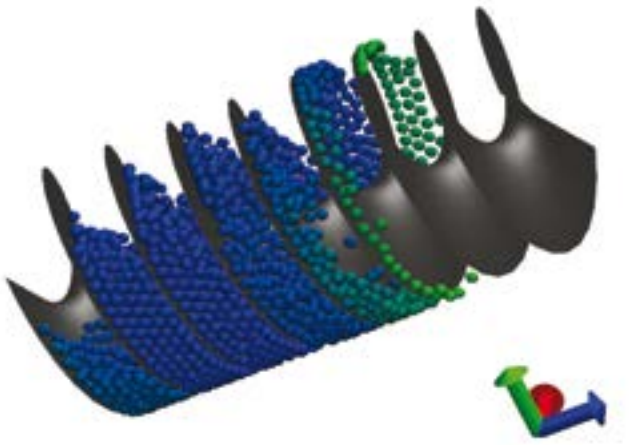

Fig. 6 - Snapshot of the screw conveyor simulation coloured by particle velocity. This transports the particles trough the tube. Colours indicate particle speed.

even rotation of the screw, if the user calls the move method. The screw is defined by a length, a maximum radius, the total number of twists, and a blade thickness. The mathematics of the definition of the surface will be omitted here, but they can be found in the full documentation of the code.

To check for collisions between a particle and the screw we have to find the point on the screw with minimum distance to the particle. This minimum is not analytically defined and Newton's method is used to quickly iterate to it. We then use this minimum to check if the screw and particle are in contact. Again, full details of this process can be found in the documentation of the code. The important points are: this process is invisible to the user; it is quicker than the triangulation method (if more than a hand full of triangles are used); it is more accurate, even in the limit of a large number of triangles; and, this method has no artificial numerical constants i.e. the number of triangles used for approximation of the actual screw.

Two standard industrial applications are used to illustrate the screw: Figure 6 shows an example of a screw conveyor, in this case, as the screw turns and particles are transported along its length, i.e., from left to right. In industry, screw conveyors are often used to transport particles to the next processing step. Figure 7 shows a screw feeder simulation, where the screw is positioned in a box, with a circular tube attached at the front end. The purpose of the feeder is to push the particles from the container into the tube to possibly feed another machine. Industrial apparatus simulations are able to provide detailed information on the flow inside the machine, which are difficult to obtain from experiments. With this detailed information one is able to investigate the optimization of these processes.

Parallel Mercury-DPM (A. te Voortwis)

As the number of particles in a system increases it becomes unavoidable to solve the problem in a parallel manner. Therefore a parallel version of Mercury-DPM is currently under development. The implementation consists of a spatial domain decomposition in which the simulation domain is split up into several smaller domains, each of which is simulated in a separate process, such that each process can be seen as a 'standalone' simulation. This approach allows for the parallel implementation to be very transparent; it is simply a layer between the driver-codes and the kernel. The necessary communication (i.e. particles moving from one domain to another) between the different domains is done through the MPI protocol, and the communication overhead is minimized by ensuring that, in general, domains only communicate with their direct neighbors.

This approach ensures the proper scaling of the performance with an increasing number of processors. The bottleneck in this implementation would be the output of the result data, since this traditionally requires the sending of all data to a single process which writes the output. To overcome this issue, the HDF5 binary file-format is used for the file output because the library of this file-format allows each process to write its data in parallel, minimizing the overhead. The combination of the flexible spatial domain decomposition and the parallel file-output ensures that Mercury-DPM scales very well from an average desktop PC up to large scale parallel high performance computer systems. With the introduction of the new HDF5 format, also the serial code has to undergo some significant changes. For this reason, the launch of the parallel version is due for the next major release of Mercury- DPM.

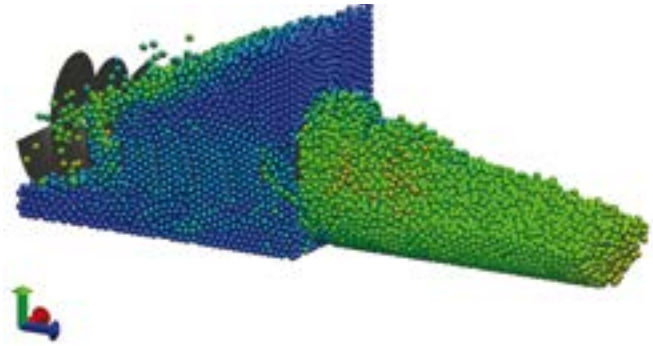

Fig. 7 - Snapshot of the screw feeder simulation coloured by particle velocity. The screw pushes the particles out of the box into the tube. Colours indicate particle speed.

If you are interested in Mercury-DPM?

Hopefully, by now you are interested in trying out Mercury-DPM for yourself. If you would like more information about the code, it can be found on the Mercury-DPM website http://www2.msm.ctw.utwente.nl/MercuryDPM. Alternatively, you can obtain updates and information about the code by joining the mailing list. To do so, simply send an e-mail to listserv@lists.utwente.nl with subject: subscribe and Body: MERCURY-USER <your full name>. This is a low volume mailing list and typically you will receive no more than one e-mail a month. The code itself is available for a public svn repository and details of how to obtain and install it can also be found on the website.

Mercury-DPM was originally started as a research code at the University of Twente, to meet a local need for a tool that was not available in existing simulation codes. Since then it has grown and gained a few tens of external users, until now purely by word

of mouth. Therefore, we have decided to make an official public release of the code, which will coincide with the publication of this article.

The Mercury-DPM release philosophy and guarantees to users

Mercury-DPM is an actively developed open-source scientific research tool, which works on a kernel and driver pattern. Some of the authors have used these types of packages before and have often run into the problem to spend time developing the driver, and then a new version of the kernel comes out and nothing works anymore. Then you have to spend time rewriting your driver to get back to a square one.
We are already quite happy with our interfaces in Mercury-DPM and expect them to change very little in the future. However, we will also give the following two guarantees. Any driver code written for version 1.x will work in each version 1 and 2 kernels. New interfaces and modifications to interfaces will initially be introduced in parallel to the old interfaces. The use of an old interface will throw a warning to the users that the interface is to be withdrawn in the next major kernel update and will explain how to convert to the new improved interface. Secondly, there will not be more than one major kernel update per year. This means that any driver code written in the current version of the kernel is guaranteed to work in all new versions for the next two years, at least. Moreover, if after every major kernel update, i.e. once a year, you spend a little time responding to the warning your code generates, it will always work in the future version of the Mercury-DPM kernel.

Mercury-DPM is still actively developed, and we have many grand plans for future features and extensions. These include added smooth particle hydrodynamics, direct coupling with continuum solvers, a graphics interface to aid ease of use, etc. Finally, if Mercury-DPM does not have a certain feature you need, we are always open to collaborate and to add such a feature. Actually, some of our current features arose in exactly this fashion; for example, the helical rotating screw wall.

Code development acknowledgement

Mercury-DPM was started by Anthony Thornton and Thomas Weinhart, and is currently actively developed by Thomas Weinhart, Anthony Thornton and Dinant Krijgsman, with input from Stefan Luding and Onno Bokhove. Stefan Luding provided a complete working particle simulation code with a state-of-the-art set of contact models that has been used as both a validation tool and as a reference guide for various features of Mercury-DPM. Additionally he has provided a great amount of theoretical and technical support in the area of (advanced) contact laws for granular materials and coarse graining. Rudi Fransen developed the current support for visualizing the output of the code using VMD. Ate te Voortwis is currently working on the parallel distribution of Mercury-DEM.

Financial Support

The development of Mercury-DPM has benefited from financial support provided by grants primarily obtained by Stefan Luding and Onno Bokhove. A full list of the grants that have (in part) supported the development is:

(1) the late Institute of Mechanics, Processes and Control, Twente (IMPACT) as part of the research program “Superdispersed multiphase flows”;
(2) STW project 11039.
(3) NWO VICI grant 10828;
(4) DFG project SPP1482 B12;
(5) FOM project 07PGM27
(6) STW MuST project 10120.

We would like to thank all parties for the essential financial support they have provided to this project.

Anthony Thornton, Dinant Krijgsman, Rudi Fransen, Sebastian Gonzalez, Deepak Tunuguntla, Ate te Voortwis, Stefan Luding, Onno Bokhove, Thomas Weinhart