EPCC-SS97-05

The Discrete Element Modeller 2

Timotej Ećimović

Abstract

The Discrete Element Method (DEM) is a modern computational method for simulating granular material. The material generally consists of many particles and as contact forces are used to calculate the interaction between the particles this turns out to be a very numerical demanding task. A serial version of the DEM was parallelised as an SSP project in 1995 [5]. A visualiser was subsequently added to this code to perform a run time visualisation of the system. The visualiser was *tightly coupled* to the existing part of the numerical code. The aim of this project is to upgrade the existing software and use a new programming paradigm: the visualiser is decoupled from the numerical core and communications between the two is now done using TCP/IP. In this document the existing structure of the code is outlined and the approach taken for the new software layout is described. Finally this new setup is used to simulate the shear-box problem which models the frictional forces between two boxes filled with sand and is of interest to Civil Engineers.

Contents

1 Introduction

One of the main purposes of this project was to decouple a run-time visualiser from a numerical core of a Discrete Element Modeller, and to use the decoupled program in a simulation of a shear box.

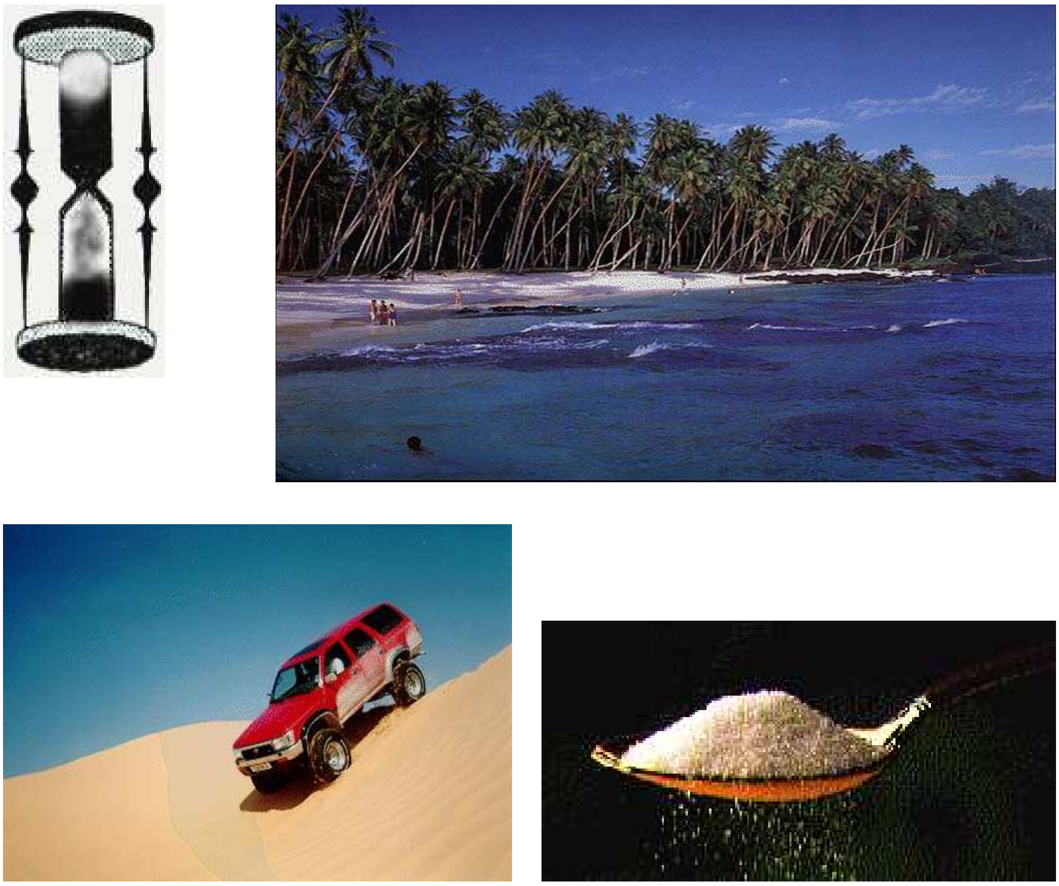


Figure 1: The kind of phenomena the DEM model is concerned with.

The Discrete Element Modeller is a program, which adopts the principles of the Discrete Element Method [4], to simulate the properties of a granular material. Granular materials appear in many real-life phenomena and applications are shown in figure 1. Every particle in the simulation is treated as a finite size ball. Interaction forces are short-ranged, and are applied only to the particles that are in direct contact. The types of forces used are: repulsive contact forces and frictional forces between particles. The forces are calculated at every time-step and the equations of motion are integrated to obtain the dynamical history of the simulated system.

The program was originally parallelised by an SSP student in 1995 [5]. The parallelism is implemented using a regular domain decomposition as shown in figure 2. A Motif/OpenGL based visualiser was later added by EPCC staff as part of a demonstration package for Digital at SC'95 [1]. The visualiser was constructed using the X-designer GUI building tool. It was tightly coupled to the numerical core at the source level. One of the processors was dedicated to run the visualiser. MPI (Message Passing Interface [2]) was used to communicate information between the visualiser and the processors running the numerical core. All the numerical part of

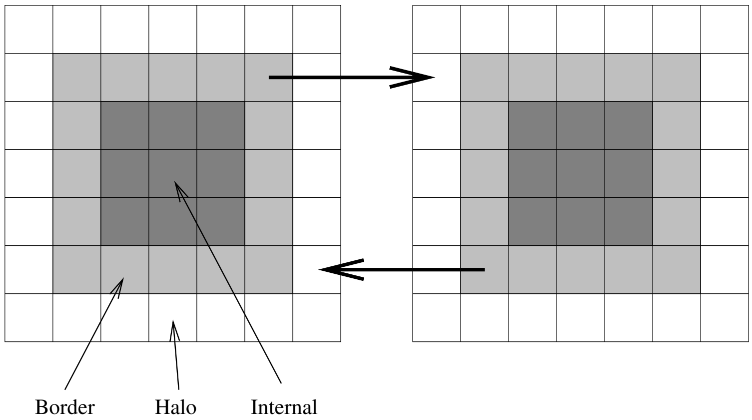


Figure 2: Halo exchange in existing parallelisation model.

the code was written in Fortran 77 and the visualiser in C.

The decoupling of the visualiser from the core numerical part was desirable in order to be able to run the simulation across a greater range of parallel architectures including those that did not have access to visualising libraries at the processor level, *e.g.* the Cray T3D at EPCC.

1.1 Definition of the Problem

In the new decoupled setup sockets are used to communicate between the numerical core and the visualiser. MPI is still used within the numerical core processors. An existing socket library was provided but this did not prove to be useful for the purposes of this problem as the library was at the same level as the UNIX system calls and insufficient error detection was provided. The UNIX system calls were used directly instead and a higher level library was created.

A visualiser to take into account the new communication capabilities had to be implemented. One possibility was to upgrade the existing visualiser, which was Motif/OpenGL based. However as the more flexible software layout with easier upgrading, was desired by the author, a completely new visualiser was created instead. This was based on the Tcl language, using the Tk toolkit for the user interface [3]. A shared library with socket communication stubs was created with Tcl wrappers provided, and a new widget was created, specially designed for the purposes of the DEM visualiser.

2 The Code

This section describes the state of the code at the beginning of the project. A careful examination of the code was necessary for a successful decoupling and the information thus found might be useful for future programmers as there was insufficient documentation. Those parts relevant to any further decoupling work, have been described.

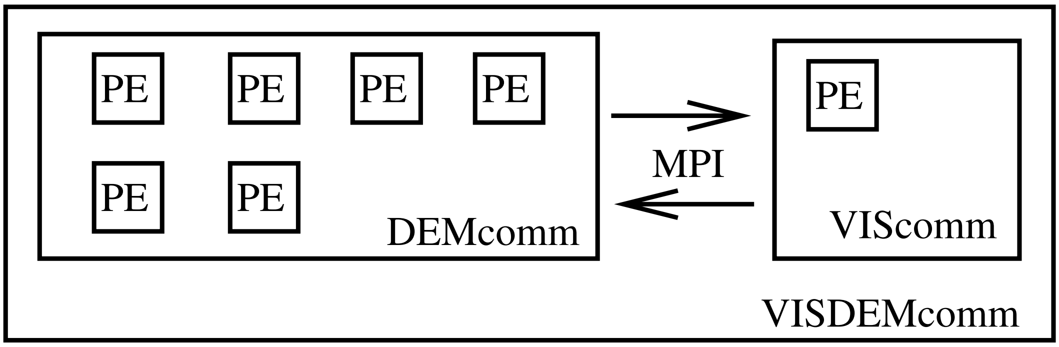


Figure 3: The parallelisation strategy adopted in the original code.

2.1 Background

A large amount of time was dedicated to reading and understanding the existing numerical code. The original documentation had either been lost or never made properly. The original author was untraceable, as was the SSP student that parallelised it.

Since the main purpose was to decouple the code, much more attention was payed to the parts of the code that deal with communication between the visualiser and the numerical core. However, since much work has also been done examining the data-structures in the numerical simulation these will be described as well.

2.2 The source layout

The code is in two main parts: the visualiser and the numerical core. The first is written in C, using X-designer as a tool for creating the user interface. The second is a pure Fortran 77 code. The `main` program is contained within the Fortran part. The C entry point is made through the `vis_main()` function. The code in its current state has preprocessors directives (#ifdef mechanism) for compiling it into two executables instead of one should the machine not support mixed language programming.

The communications are done using several MPI communicators. Two new communicators are created: VISComm and DEMComm. These two are internal communicators used for communications in the visualiser and the numerical part of the code respectively. The structure of this setup is schematically represented in figure 3.

For the purpose of the decoupling, the most important feature is the inter-communicator VISDEMComm, which is created to effectuate communications between the numerical part of the code and the visualiser. The process with rank 0 within the DEMComm communicator is responsible for talking to the visualiser. The visualiser only uses one processor, so communications within VISComm does not have any meaning.

The tags used in communications within the DEM numerical core are an important fact to note for the future programmers. They are in a file `common_tag.h`, and the newly added facility should not use the same tags.

2.3 The numerical core

The numerical core of the program is contained in six source files. The inter-relation of the procedures in the code is summarised in figure 4. A capital F extension is used as these files require to be pre-processed before being compiled. Briefly within the context of the decoupling of the codes we have:

comm.F: Contains functions, which are used in the communication between the calculating processes. No calculation is done within the visualiser process, so this part is not important for the decoupling of the codes.

contact.F: Contains functions for calculation of the contact forces used in the simulation. There are no communications within this source file.

datstruk.F: Contains utility functions for manipulating data structures. These represent the cells in the simulation area which are used to create and work on the linked-lists and deal with the corresponding arrays. No communications are contained within this part of the code.

integ.F: Contains integration functions used to update the particle and wall positions.

io.F: This file contains no communication code either. It does contain functions which are used in the processing of initialisation files. There are some MPI calls, but only for communications within DEMComm. This file however is important for subsequent work, as it defines the format of the input files.

main.F: All the communication with the visualiser is done from the functions contained within this file. The main block calls the visualiser initialisation (`vis_main()`) and creates inter-communicators. The leader process of the numerical core uses the VISDEMComm inter-communicator to initialise the visualiser with the data obtained from the input files. This is the first point of code where major rewriting is necessary. The other two main points are the `send_state` subroutine, which deals with sending all the numerical data to the visualiser and the `need_update` function which reads a control integer sent from the visualiser and subsequently performs the requested action. The protocol is very simple: a value of 1 requires the visualiser to pause in the computation, 2 makes it resume and 999 will trigger a shutdown and the program will exit. A value of 0 is also used as a request for an update from the visualiser. The last communication that is done consists of sending of -1 to the visualiser which is a signal indicating that the numerical part of the code is about the exit and the visualiser should exit as well.

def2.com: All the global variables are contained within this file, which is included by every subroutine in the numerical core.

2.4 Problems with the numerical core

It was found that the existing numerical code had a bug. This caused it to crash at various points in the code. The reason for this being that an array index goes out of bounds producing a segmentation violation. It was observed, that code works fine when only one processor is used thus the problem appears to lie in the parallelisation code. There is also some evidence of previous bug hunts: parts of the code have whole blocks commented out. These blocks were

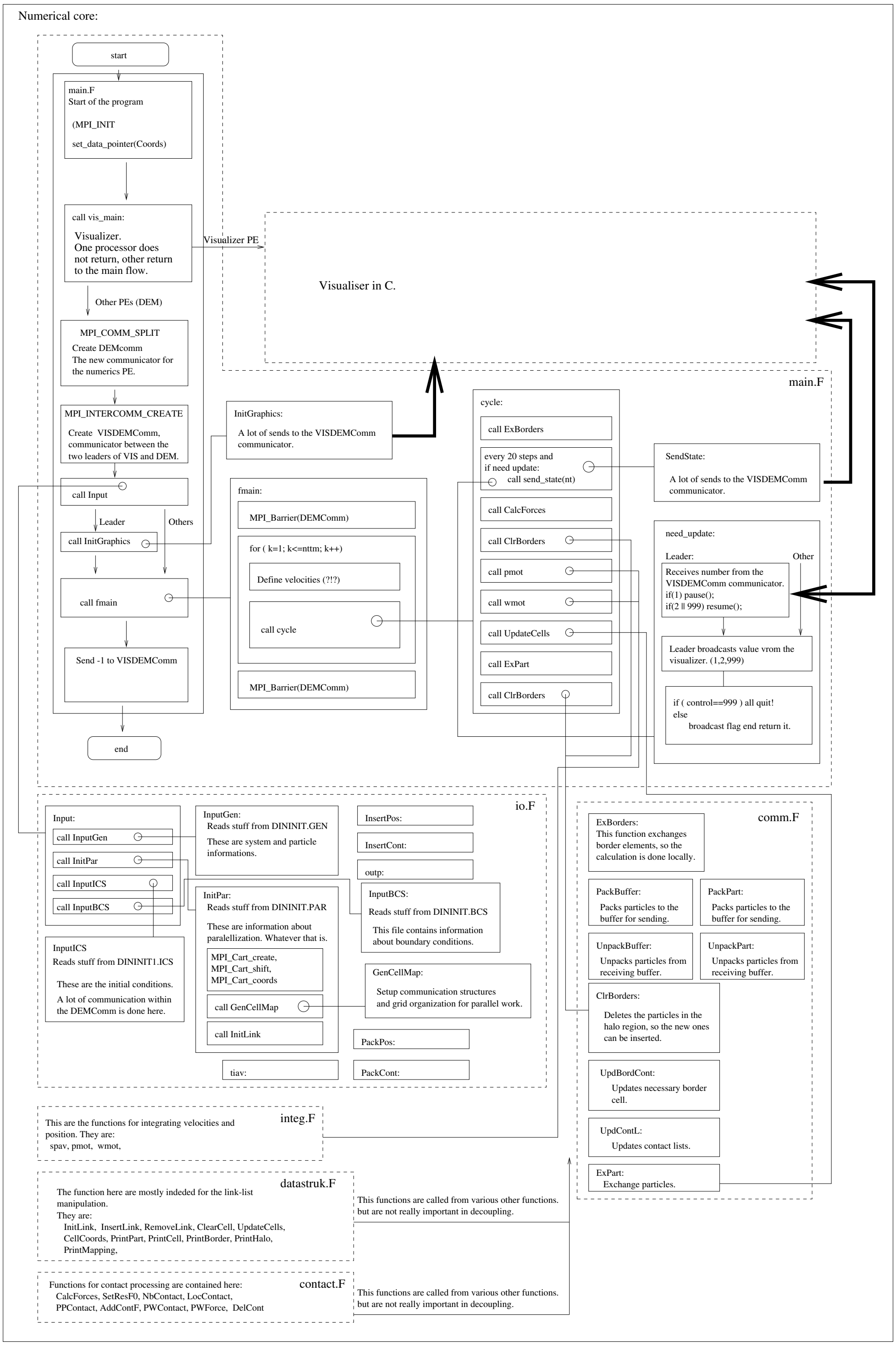


Figure 4: Schematic structure of the numerical core.

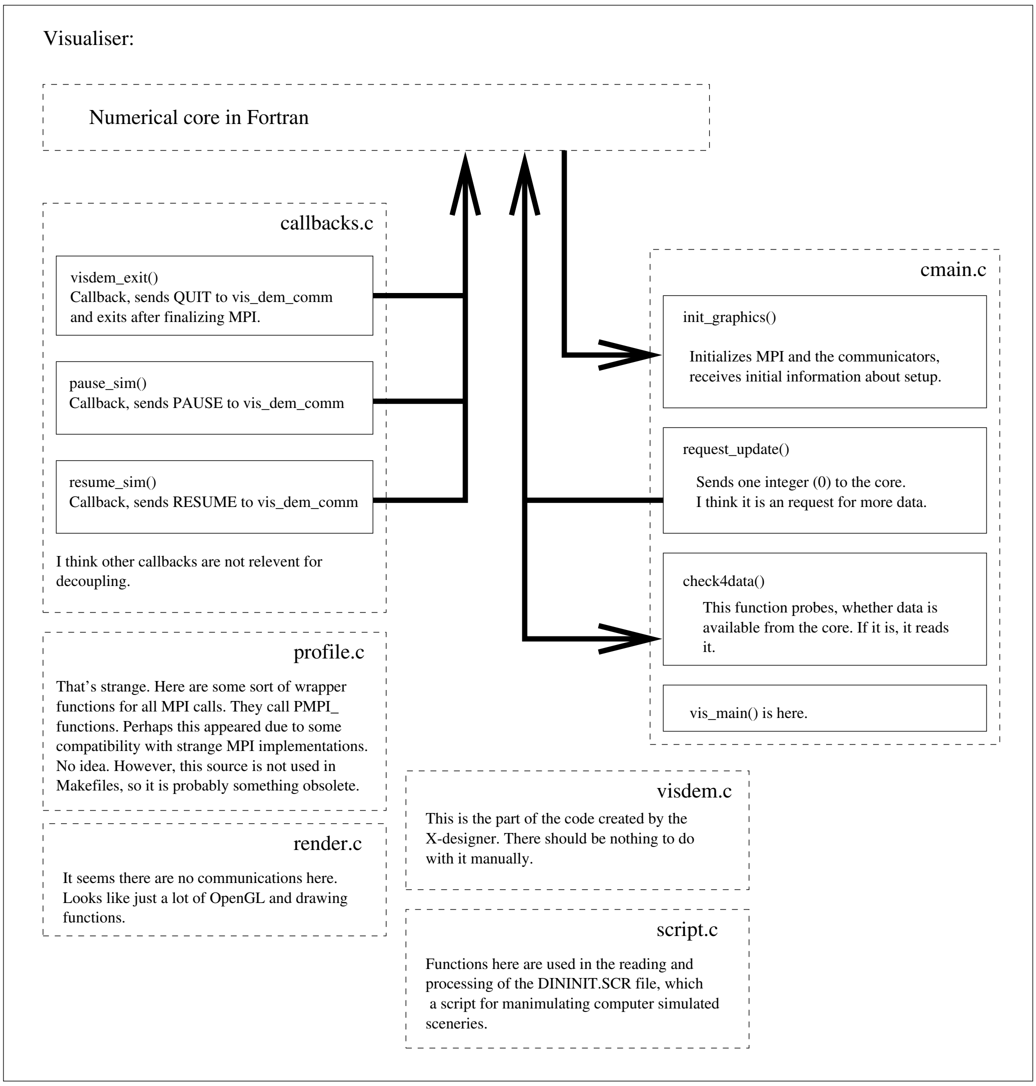


Figure 5: Structure of the visualiser.

present in case some critical error was found causing the code to abort. The reason for removing the critical error blocks was traced back to the work when visualiser was added to the code. The programmers, who were preparing the code for the demonstration realised that critical error condition were satisfied as some point, and the program would abort. But for demonstration purposes a continuous run was mandatory so these critical error blocks were taken out of the code.

If the numerical part of the code was compiled with the `-C` flag, which causes the run-time checking for out-of-bounds indices, the code crashed at different time-steps but very early on in the simulation. This definitely points proved that there is a problem with the parallel part of the code. An attempt was made to trace the error but it could not be detected exactly. The problems appeared with the linked-lists of the particles within the cells which were corrupted with the invalid particle references, but the real reason for this behaviour was not found.

2.5 The visualiser

The visualiser code, written in C, is also contained within six files. However one of them (profile.c) is only used for profiling purposes. A graphical representation of the structure of the code is given in figure 5. A macro SEPARATE_EXEC could be defined for compilation of the visualiser and numerical core to compile into two separate executables as has been mentioned earlier in this report. A brief description of the source files is given:

callbacks.c: This source file contains widget callbacks, that are activated by clicking and dragging the mouse pointer around the visualiser. Only three are relevant: `visdem_exit()` is the callback for the exit button, `pause_sim()` is the callback for the pause button which sends PAUSE (1) across the `vis_dem_comm` inter-communicator and finally, `resume_sim()` which sends RESUME (2) and is the callback for the resume button.

cmain.c: Three relevant function are in this source file. The first one is `init_graphics()` routine which receives all the initial setup information, so it can plot the scenery. The second is the `request_update()` function which send an integer 0 to the numerical part of the code. The third function, `check4data()`, probes whether data from the numerical part is available for reading and if it is, it reads it.

visdem.c: This is code created by the X-designer. Nothing of interest here.

script.c: Contains functions which deal with the reading and executing of the internal scripts. A user can create scripts, which contain the information about any action that will take place in the scenery.

profile.c: Used for profiling the code. It is not linked into the final executable at all.

render.c: This file contains the OpenGL drawing routines.

Some strange behaviour was observed in the original visualiser. First there is a colormap problem. It seems that the drawing widget requests a new colormap for its use, though it does not need it at all. This problem can be solved by explicitly setting the `installColormap` resource to FALSE as the default setting is TRUE. This has been hard-coded into the code, but it can also be done via the resources file as well, without recompilation.

The second problem appeared with the MesaGL functions. On the CS2 platform, it has been observed, that the `glXSwapBuffer()` function crashes at some point, the reason being that the OpenGL contexts for the legend and the drawing area get interchanged. Eventually swapping the buffers produces a fatal error as both these areas are of different sizes.

During the project, a new visualiser was created from scratch. Therefore the information about the old visualiser is no longer important for further work. One of the reasons for the new visualiser was the development time, which was limited. Since Tcl/Tk is the GUI building toolkit with the shortest development time span, as believed by the author, a decision has been made to use that highly flexible and extendible language instead of Motif/OpenGL combination. Other reason was the fact, that most of the problems of the original visualiser could be traced back to the OpenGL routines, which all in all are a tool created for much more complex drawing, than is done in the DEM visualiser.

3 Decoupling the Visualiser and the core application

3.1 The guidelines for the development

The decoupling was performed at the points described above. There should be no MPI calls whatsoever in the visualiser, and the Fortran numerical core should only use MPI calls for internal communications. The new model is shown in figure 6.

Since the uninterrupted flow of the numerical core is the main objective, one of the foreseen problems might be the response to events from the socket interface. One possibility with sockets, is to install a signal handler for any event on the socket. The side-effect is the introduction of a global mechanism for dealing with the events, and the irregular flow of the program.

The implementation in the old version checks the MPI buffers after every 20 time steps, and then performs any necessary actions requested from the visualiser, *e.g.* sending new data to it. The new signal method might prove more useful, since the numerical core spends more time on the computation when no user interaction is necessary. In this case, all other actions other than the computation itself, would be initiated by the visualiser. It could easily be sent into 'silent mode' and would thus not use any CPU time to communicate with the visualiser.

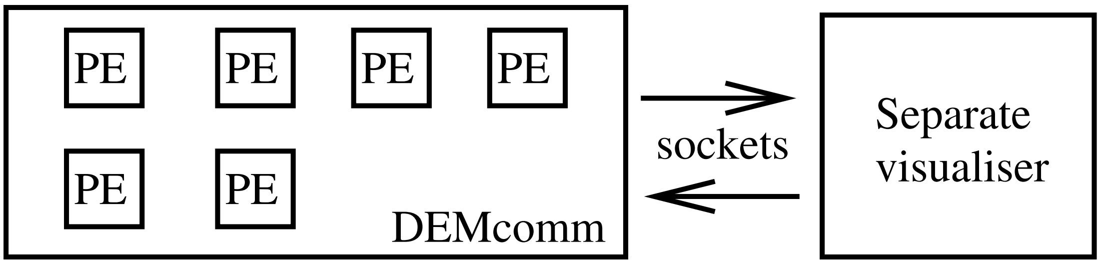


Figure 6: New model of communication.

3.2 Visualiser-numerical core communication

The model adopted was the one where the numerical code itself would also act as the socket server [6]. Some consideration was given to the possibility of having a dedicated PE which would take care of all the socket serving and poll for data from the other processors. This would lead to inactive PE even in the cases where no socket communication was required. The other possibility, which was chosen for this implementation, is one where all the processors are involved in the numerical calculations, but one of them (rank 0) also serves the socket. This leads to a slight load imbalance but in cases where there is no visualiser connected the imbalance vanishes. This is why this model was adopted. A schematic representation of this is shown in figure 7. The code now would still be able to run on the Cray T3D as fast as possible, without any visualiser ever being connected to it.

The process with rank 0, which acts as the server, implements a signal handler for the `SIGPOLL` event (`SIGIO` under BSD). This signal handler will set a flag which will cause the event handler to be called from within the main calculation loop. In those cases, when no events are pending, the extra work imposed on a serving processor will be just a test, whether flag equals 0 or not. This work can be neglected, compared to the load-imbalance of the problem itself.

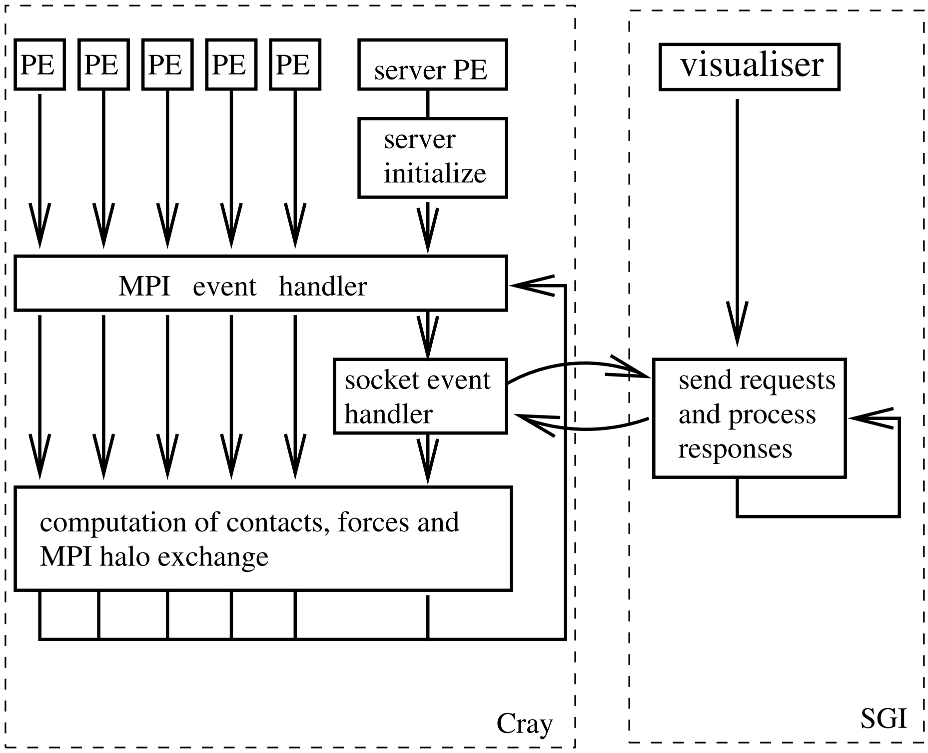


Figure 7: The new layout.

On initialisation a listening socket is created and added to a linked-list of file descriptors which holds all the socket information. They can be either servers or clients. Usually there will be only one client and only one server socket, but the case of many clients (visualisers in our case) being connected to the code at once is possible. The implementation of this is not any more complicated than the implementation of a single client. There is no reason why there should be an explicitly imposed limitation of only one visualiser that can be used.

When the event flag is set to a non-zero value by the signal handling function, the event handler is called and `select()` [1] system call is activated in a non-blocking mode. The server and all the existing client sockets are checked for a read condition. If the listening server socket detects data it means that a new client is trying to connect and the server `accept()`s it and adds the newly obtained file descriptor to the linked list as a new client process. If, on the other hand, a read event is detected on a client socket, the data read is represented as a DEM protocol request and the server tries to respond to it.

All the socket operations are done in a non-blocking manner. On creation, sockets are set for SIGPOLL events and to non-blocking mode with `fcntl()` and `ioctl()`. Reading is done into a buffer, which is local for each socket. Initially 1 Kb is allocated for the buffer, and if needed this buffer can be enlarged. After the event is handled, the buffer always shrinks back to the original size of 1 Kb, so there should not be any memory leaks. Anyway, at this stage of the project, it is improbable that requests larger than 1Kb will be generated at all.

[1] Although `select()` is a BSD feature, and other part of the code is mostly System V compliant, I think it is common enough on all SYS-V platforms as well, so we do not really need SYS-V `poll()` function.

3.3 Communications within the numerical code

The communications which deal with the event handling are discussed in this section, not those concerned with the computational data exchange.

All event data is exchanged using the `DEM_EVENT_TAG` tag, which is set to 237. There was no particular reason for this choice.

MPI events are generated by the socket-controlling processor, if the request from the visualiser is such, that it requests the cooperation from all processes. The communication mode is a blocking buffered communication, so the initial buffer of 8 K-bytes is attached to the MPI. This should do, because the protocol consists of a two integer numbers. First integer represents the event number, and second is the file descriptor of the socket to which results are to be send. Second is needed, because MPI event handler sends the results back by itself. The reason for this is the fact, that the reverse events for the socket controlling process would unnecessarily complicate the code. Therefore the socket number is added to the event number. Of course, only process with rank 0 should do anything with this file-descriptor, since other processes don't have access to the socket.

Interesting point to note is the fact, that the MPI event processing is done before the socket events are processed later in the loop. Although it seemed reasonable to do the event checking after the socket events are processed, this proved bad, because all the other processors, which didn't have to deal with the socket usually passed the part where MPI is probed for the events. However the processor with rank 0 did not, so the controlling processor actually reached the MPI event handling loop, when all the other were already involved with the numerical computation. If the event processing requested some synchronous operation, this would lead to possible deadlock. Synchronising the processors in a MPI event processing routines, would cause all the processes wait for the controlling one, even if the socket handler requires no other MPI events.

Therefore the MPI event checking is done after the computation of a single time-step takes place and before the sockets are probed for the events. Since the numerical part of the loop loosely synchronises the processes, this should lead to all the processes reaching the MPI event checking at a time, which would depend upon the number of computations they have to do. If the controlling processor is assigned the smallest domain, the time lost during the socket processing could be gained back this way.

The library created for the purpose of decoupling is written entirely in C and full documentation for it is provided in appendix B on page 23. It features reliable communication across sockets and should be UNIX platform independent. It imposes no limitation on number of clients connected and on the size of requests or responses.

3.4 DEM protocol

DEM protocol defines the communication between the visualiser and a client. Its structure is shown in the figure 8. The major and minor numbers uniquely define the request or response, and other data is used when needed. In some cases, the 2-byte value is enough (reporting the number of processes), but when larger amount of data is transmitted the XDR (External Data Representation) has been used. This format was developed for the Sun Remote Procedure Call library, and is standard on all UNIX platforms. It was used because its purpose is exactly the functionality that we require, that is representation of data in a platform independent manner.

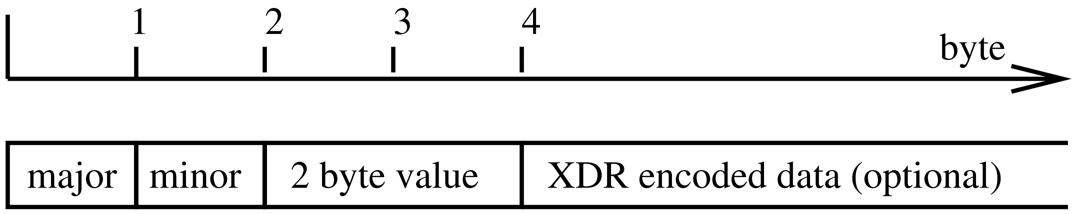


Figure 8: DEM protocol.

The implemented responds and requests are documented in appendix A on page 21.

3.5 Requests from the socket server PE to other processors.

Communications within numerical core are also necessary, to adequately respond to the requests from the visualiser. Upon any visualiser request, that requires interaction from all processors, a message is broadcasted to all processors (via MPI), which contains internal MPI protocol request. Upon receiving this message, processors will send the requested data to the controlling processor, which will then send it further to the visualiser.

The implemented internal MPI requests are listed in appendix A.2 on page 22.

3.6 Future work with the library

A necessity of the library is the dynamic installation of request handlers. In present state, the request handlers are contained within the library, but these are application specific, so they really shouldn't be there. Therefore the most important work in the future should be a mechanism for dynamic registration of request/response pairs and the corresponding handling functions for the requests. A mechanism should also allow the registration of either Fortran or C functions for this purpose.

Other feature, which should be implemented in the future is a proper implementation of security issues. Protocol requests should be separated in groups, according to their availability, and password authentication scheme should be created, to disable access to the numerical code from the unauthorised client. Multiple level security passwords should also enable certain clients to obtain the data from the numerical core, but disable them from stopping or aborting the numerical run.

3.7 Other issues regarding the numerical core

Some work has been done on the numerical core, which was not directly connected with the decoupling. The use of the socket library was throughout the development regarded as an option therefore certain preprocessor macros are used to control this behaviour.[2]

SOCKETS : If this macro is undefined, the code will not link with the socket functions and will have no network ability. It will be just the original Fortran numerical code.

[2]The author of the project tried very hard to keep this behaviour as it is described here, but it has not been thoroughly tested. The future user, whose work should rely on these properties should verify the proper behaviour.

DECOUPLED : This macro causes all the original coupled code to be disabled. If you don't use DECOUPLED and don't use SOCKETS, the code should compile into one executable with the visualiser as before. However, you should rather use the original distribution for this purpose.

One feature added to the numerical core was the run-time computation of the stresses in a certain region. The theoretical concepts shall be discussed in a section 5, only the programming issues shall be documented here. The new input file `DININIT.STRIP` has been defined for this purpose (as documented in appendix C). The functions for reading this file have been added to the `io.F`. A new source file has been created, named `forces.F`. It contains subroutine called `InternalStress`, which takes three double precision numbers as parameters, calculates average σ_{xx}, σ_{yy} and σ_{xy} in the strip region and stores them in the passed variables correspondingly.

The function is called by each processor and MPI communication is used to do global sums of the results. Please, note that the code has been tested mostly in serial, due to the problems with the parallelism, mentioned in section 2.4.

4 The new visualiser

The new visualiser has been created using Tcl language with the Tk toolkit [3] for the user interface. This tool has been selected among other possibilities (Motif, X-designer), because it has probably the fastest development time. At run-time the code is slightly slower than Motif, but not slower enough to cause inconvenience to user.

The communication facilities are provided to the Tcl code via shared loadable module, and visualisation is done using a newly-created DEM visualising widget. The latter was developed because standard Tk drawing canvas is too slow for the large amount of data.

If a target platform does not enable shared module loading, the old mechanism for static loading should be use to obtain a new Tcl/Tk shell with the added capabilities.

Tcl/Tk is in public domain, available on many anonymous FTP sites and compiles on every UNIX platform.

4.1 Loadable shared module

The communication stubs for the new visualiser have been put into the shared object, prepared to be loaded by the Tcl language, which is the base for new visualiser. The functionality of the new module and the new functions available to the future programmers are listed and documented in appendix E.

4.2 Functionality of the visualiser

Visualiser enables user to connect to the remote server and get the data from it. It is not very robust in its implementation, since its functionality does not affect the numerical run. Therefore, network problems might cause it to hang and keep waiting for non-received data indefinitely. This malfunction is caused in the Tcl/Tk code and

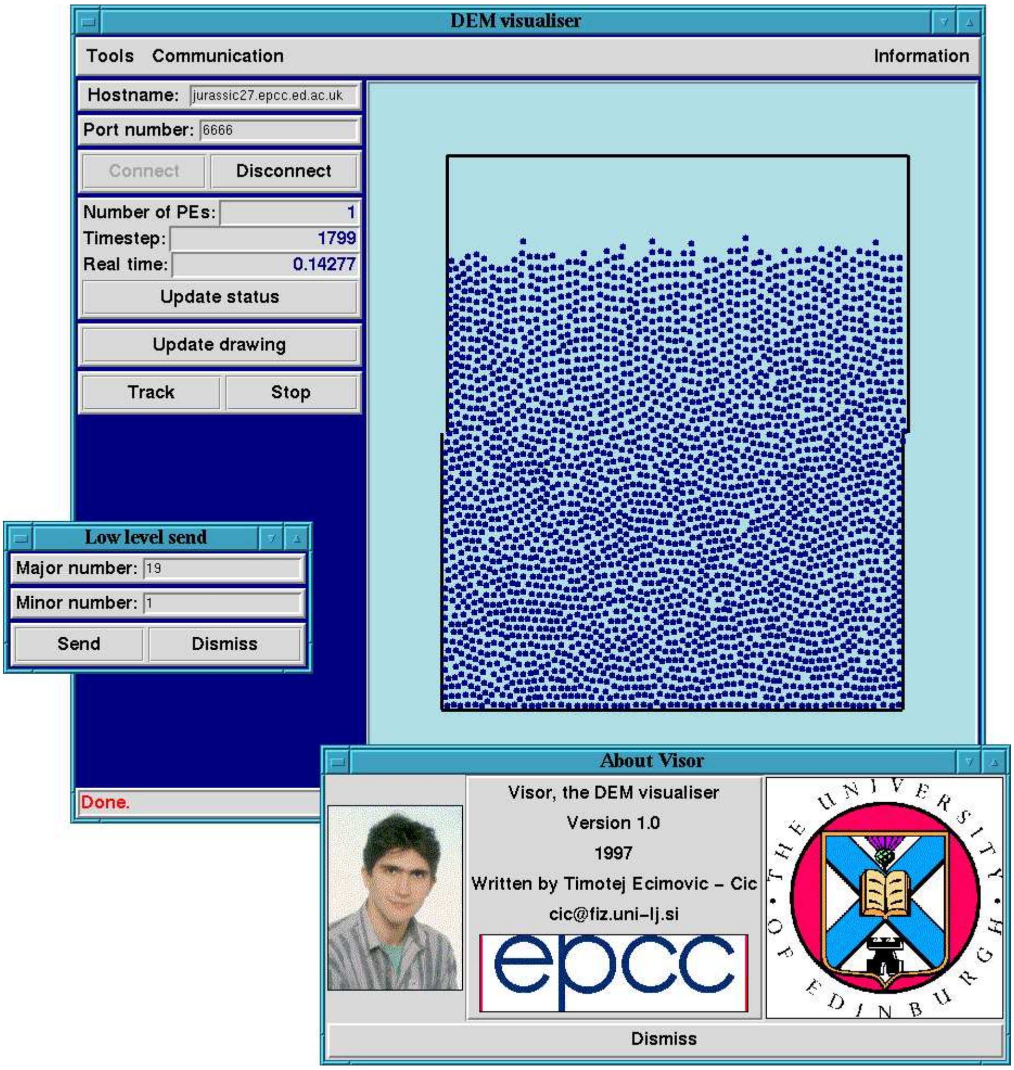


Figure 9: The new visualiser.

5 Simulations

5.1 Shear box setup

The experiment for which the new setup has been used, is the shear-box experiment. It consists of two boxes filled with sand as seen in figure 10. A certain region is required in the contact area, and then top box slides across the bottom one, and the friction force between both boxes is the desired result.

The calculation of the shear-box setup is performed in two steps. A vertical velocity downward is applied to the top wall, it's value being 0.1. While the downward velocity is applied, the stress tensor is calculated within the defined strip. As the pressure (σ_{yy}) reach certain predefined value, the downward velocity of the topmost wall vanishes, and a velocity in horizontal direction is applied to the top box. As that is happening, a σ_{xy} is being recorded, as it is proportional to the frictional force.

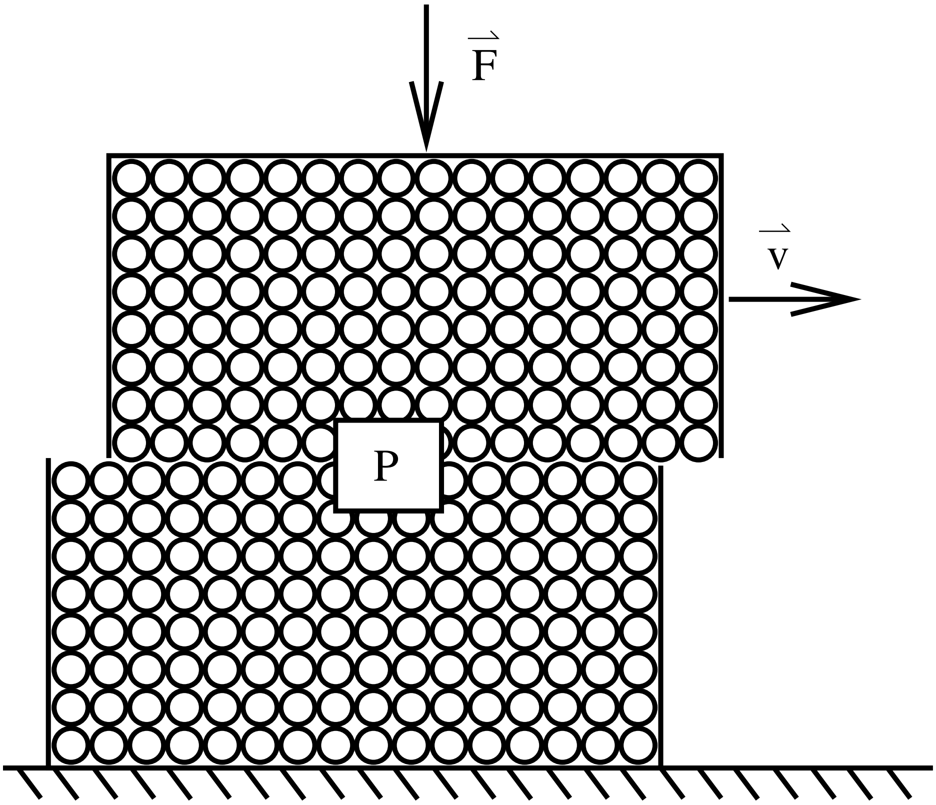


Figure 10: Shear box setup.

5.2 Internal stress

Internal stresses are calculated via Fortran function `InternalStress` which is contained in file `forces.F`. The algorithm uses the formula

$$\sigma_{ij} = \frac{1}{V} \sum F_i L_j$$

where V is the total area of the strip, F_i is the contact force component parallel to the i-th axis, and L_j is the j-th component of the vector, joining the two particles.

The contact is considered in the calculation if both particles are within the specified strip coordinates. The information about the coordinates is stored in the input file `DININIT.STRIP`, which is read upon startup by the function `InputStrip`, in `io.F`.

Please note that for the results obtained in this report, the area of the strip actually was the area of the box, included in the strip. But this code is not general, so it should be removed from the program, and a general mechanism for determining the area of the box within the strip should be used instead.

Every processor calculates it's share of stress, and global sum is used to sum up the results obtained by each PE.

5.3 Results

The internal pressure in the strip layer had to be set to prescribed value first. Therefore a numerical experiment has been performed, with constant downward velocity for the topmost wall. The increase in the internal pressure is seen in figure 11.

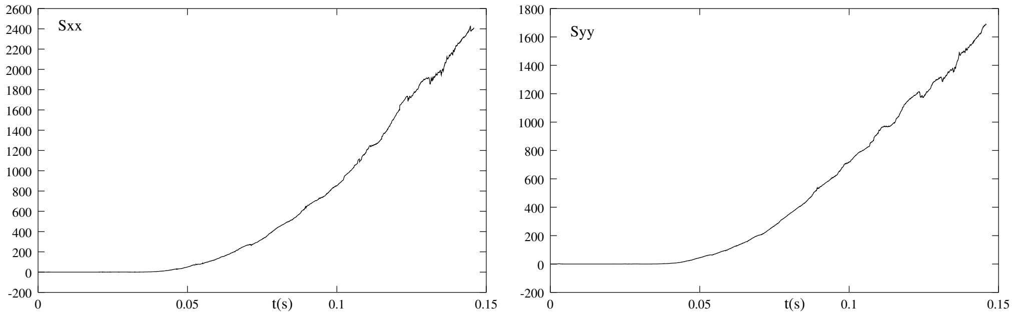


Figure 11: The internal pressure increase at constant downward velocity $v_y = -0.1m/s$.

The shape of the pressure curve is smooth enough, to use a very simple condition for the start of the horizontal velocity. As soon as a prescribed pressure is reached, the downward velocity of the wall stops, and the horizontal velocity of $0.0002m/s$ is applied to the three walls, which make up the top box.

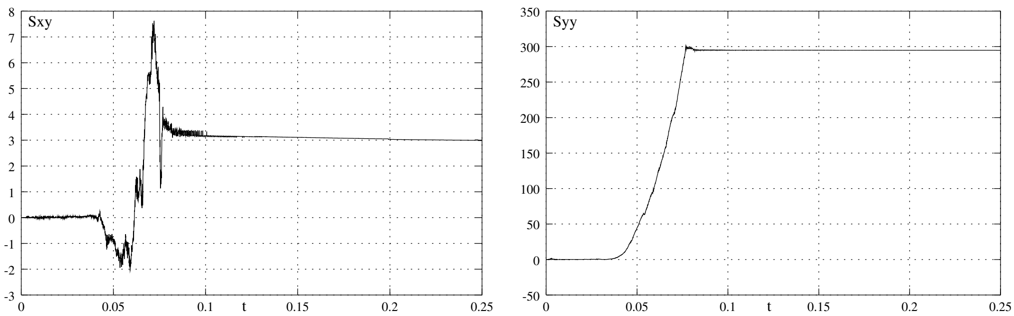


Figure 12: The plots of the σ_{yy} and σ_{xy} at pressure 300 kPa.

After the horizontal velocity is applied the shear component of the stress tensor, σ_{xy} is measured. The value is linearly proportional to the frictional force and the expected behaviour is the increase of the force to the maximum value, and then the saturation to the limiting value.

This effect can be seen in figure 12 and figure 13. It can be seen how the internal pressure increases to the point where the rightward motion starts, and the shear component curve has the shape which is expected by the predictions.

As the future work should include the experimental work, which should or should not, confirm the results obtained by the model, the results should be fully interpreted at that time.

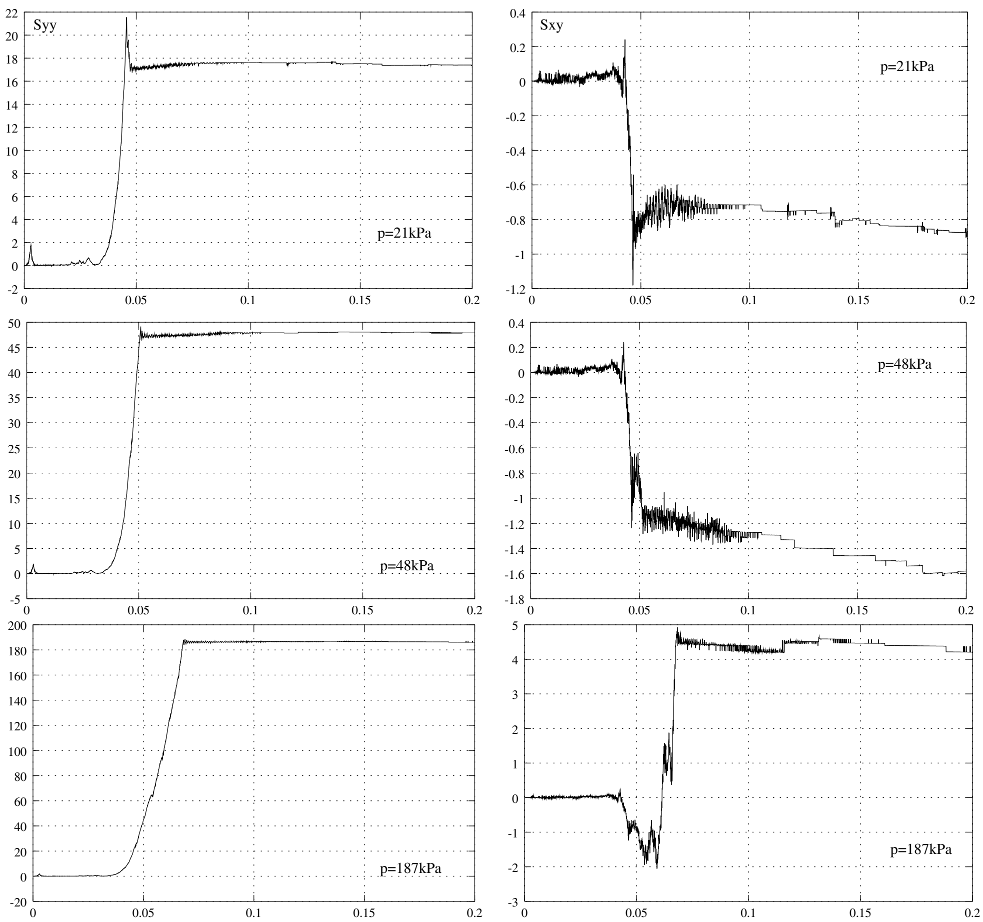


Figure 13: The plots of the σ_{yy} and σ_{xy} at different pressures.

6 Guidelines for the further work

The major part of the future work should be concerned with parallelisation. The code should be either extensively debugged, or preferably rewritten. The memory consumption in the parallelising code is deadly for the platforms with limited resources, as each processor holds the copies of the global arrays, not only the data it needs. The instability of the parallel code also makes it hard to use on large scale multiple-processor platforms, since the probability of the crash is very high. Therefore is is strongly believed, that before any major enterprise is undertaken with this code, it has to be partially rewritten using perhaps a language with higher flexibility in the area of memory management, such as C.

Some correction should also be made to the code which reads the input data, which is not very robust. Another improvement point is the generation of the output files. Currently there is only one mode available, but in the future work, hopefully more will be added.

The library for decoupling is in its early stage, and if the future work will be done on it, the major parts of improvement are the features for registration of new event-handlers. Security issues are also important, since currently there are none at all, which means that any client from any machine can connect to the numerical core and cause it to abort, which is probably undesired behaviour.

The visualiser is stable enough for use, but lacks some of the features, such as zooming or velocity display. As the Tcl/Tk code is very modular, these features can be added easily by a programmer fluent in Tcl/Tk and Xlib programming.

The widget code is written in a usually portable manner, complying to the Tcl/Tk widget creation guidelines. The code is open to any future improvements. Zooming can be done from the Tcl code itself, using Tk bindings for the mouse-dragging events. The redrawing routines in the file `redraw_widget.c` can be rewritten using OpenGL if this is desired and velocity and internal pressure displays should be added in future versions.

7 Conclusion

The aims of the project have been successfully completed. However, since the discrete element modeller itself is a continuous effort, the future development will probably take place.

The main achievement of the project, which is a decoupling, proved not only useful tool for the DEM project, but is a possible future paradigm for MPI applications. Therefore an effort has been made to create a library in such spirit. The library proved robust enough and the numerical core was not slowed down because of the new features.

The new visualiser is written in a very modular and extendible way, so any future development regarding visualising is now separate from the numerical core.

As the simulations go, the code proved unstable in parallel mode, when more than one processor was involved in calculation. Therefore the results were obtained with a single processor only. However, the newly added code is aware of the parallel environment, therefore a future project concerned with rewriting the parallelisation part of the numerical core, should not have too much work with the decoupling library itself.

The numerical results corresponded to the measured experimental values to certain point and it is believed, that the Discrete Element Modeller shall be a useful tool in the future as well.

A DEM protocol

A.1 DEM socket protocol

The DEM protocol is consisted of the requests from the visualiser which are listed in table 14 and the responds from the numerical core, which are listed in 15. The third column in the tables marks whether the requests are reserved or public. This was done as a mechanism for security, since a numerical core, which runs as a socket server is vulnerable to possible network attacks. Therefore, a mechanism was thought of, how to secure the numerical core from the network attacks. However, this feature has not yet been implemented, since it wasn't necessary for the successful completion of the project.

Major	Minor	Status	Other data	Description
13	1	R		Stop and wait for resume
13	2	R		Resume computation without stops
13	3	R		Abort all tasks (`MPI_Abort()`)
13	4	R		Request "nice" shutdown of all processes
19	1	P		Request coordinate pairs (x, y)
19	2	P		Request velocity (u, v)
19	3	P		Request angular velocity ω
19	4	P		Request all data (x, y, u, v, ω)
19	5	P		Request time-step
19	6	P		Request real time
19	7	P		Request dt
19	8	P		Request boundary information
19	9	P		Request wall information
19	10	P		Request strip information
19	11	P		Request average radius
1	0	P		Exercise all communications
1	1	P		Force MPI synchronisation
2	1	P		Request number of processors

Figure 14: DEM protocol - requests from visualiser.

Major	Minor	Status	Other data	Description
20	1	P	2 byte number	Report number of processors.
20	2	P		Numerical code is quitting.
22	1	P	1 int (rank) + 1 int (no. of points) + N double pairs	Coordinates pairs.
22	2	P	1 long (no. of points) + N double pairs	Velocity pairs.
22	3	P	1 long (no. of points) + N doubles	Angular velocities.
22	4	P	1 long (no. of points) + N quintuplets	(x, y, u, v, ω)
22	5	P	1 int	time-step
22	6	P	1 double	real time
22	7	P	1 double	dt
22	8	P	4 doubles	xmin, xmax, ymin, ymax
22	9	P	1 int + N double quintuplets	N, N x (x,y,angle,edge1,edge2)
22	10	P	int + 4 doubles	flag, x1,y1,x2,y2
22	11	P	single double	Average radius

Figure 15: DEM protocol - responds from the numerical core.

The responses usually mean a lot of data. The data should be transmitted in a network transparent manner. Characters are no problem, so the major and minor numbers (which are just two characters) are still there. If no other information is needed, than just two characters are transmitted. If response is satisfied by a short integer (16 bits) the information is stored into 3rd and 4th byte of the buffer. For example the response about the number of processors involved has the following structure:

```
buffer[0]=(char)20; /* Major number */
buffer[1]=(char)1;  /* Minor number */
buffer[2]=(char)((n/256)%256); /* MSB of the response */
buffer[3]=(char)(n%256);       /* LSB of the response */
```

This cases are in the table marked as *2 byte number* in *Other data* column. In the cases, where response is a larger pack of information, the XDR (extended data representation) functions are used as specified in `encode.c` file. Actual functions called from the rest of the code are those in `encode.c`, so it should be possible to rewrite there encoding routines without to use something else than XDR without any changes to other code.

A.2 MPI DEM protocol

The table 16 documents the implemented MPI internal requests. These are being send by the socket-controlling processor, to all processors including itself, upon receiving the socket requests which require MPI interaction with other processors.

Number	Meaning
0	Exercise connection. No response is requested.
1	Please, everyone quit NOW!
2	Stop numerics and wait for resume.
3	Resume.
4	Force MPI synchronisation.
10	Request coordinates.
11	Request velocities.
12	Request angular velocities.
13	Request all data.

Figure 16: DEM MPI protocol - requests from the controlling PE.

B Decoupling library

This section describes the decoupling library created during the process. Note that all the MPI sends are done in buffered mode, and the amount of buffer allocated is defined in file `mpi_events.h` as macro `DEM_BUFSIZE`. It should be desirable that this buffer is large enough to hold any data transmitted using MPI, since the behaviour of the code beyond this constraint has not been tested. By default it is set to hold 100000 double numbers. The code would most probably deadlock if this limit is exceeded, so the future programmers should either remove this behaviour or make sure that transmitted data is within the limit.

B.1 Macros for compilation options

Following preprocessor macros are used for controlling the compilation:

C_DEBUG : If this is defined, a lot of `fprintfs` will be added into the code. It is used for development. The macros D(), D1() and D2() are used throughout the code for printing out various debug information.

FORTRAN_APPENDS_UNDERSCORE : Macro, which causes the code to expect an additional underscore from the symbols produced by the Fortran 77 compiler.

FORTRAN_DOES_UPPERCASE : With this macro, Fortran compiler is not expected to append an underscore, but to change symbol names into all uppercase (this happens on Cray platforms). Either one of the two Fortran macros should be defined, otherwise the code will not compile well.

B.2 Description of exported symbols

DEM_send_request()

```
int DEM_send_request ( int sock, int major, int minor );
```

Defined in `client.c`.

This function sends an empty request with major and minor numbers to the socket `sock`. The request sent with this function can contain no other data but the two characters. It is intended primarily for the client use, but nothing prevents server to use it. The returned value is the value returned from system function `write()`. Upon success this function should return value 2, since that is the number of characters written to the socket.

DEM_read_response()

```
int DEM_read_response ( int sock, char **data, int *size );
```

Defined in `client.c`.

This function reads data from the socket `sock` and stores it into the location `*data`. Value `*size` should specify the size of the preallocated space in the buffer `data`, and may be zero. The size of buffer may increase dynamically and the value stored in `*size` upon return, matches the allocated space in `*data`. The programmer should therefore not be concerned with the size of the data read, since this function will allocate necessary space. If function is called with the buffer of size 0, 1024 bytes are allocated by default. The caller of the function is responsible for freeing up the memory. The socket should be set in non-blocking mode, so the function doesn't block. If socket is set in blocking mode, this function will most probably block in any case, because the way of detecting that no more data can be read from the socket is the return value from the system call `recv()`, which should be `EWOULDBLOCK` for the function to return. In case of error, -1 is returned otherwise the return value is the number of bytes read from the socket. Note that this is not necessarily the same as the size of allocated buffer, since memory is taken in chunks one kilobyte at the time.

DEM_disconnect_client()

```
void DEM_disconnect_client ( int sock );
```

Defined in `client.c`.

This function shuts down both directions of communication in socket `sock`, which shouldn't be used anymore.

DEM_connect_client()

```
int DEM_connect_client ( char *hostname, int port );
```

Defined in `client.c`.

This function attempts to connect to the socket on machine `hostname` at port `port`. If it succeeds, the socket is set to non-blocking mode and is returned. Otherwise -1 is returned as the error indicator.

decode_start()

```
void decode_start ( Decode_handle *enc, char *buffer,
                    int size );
```

Defined in `decode.c`.

Function initialises and starts the decoding process. Address at `buffer` should be evenly divisible by four, otherwise BUS error might occur. Parameter `size` should contain the size of memory available in bytes.

decode_int()

```
int decode_int ( Decode_handle *enc, int *a );
```

Defined in `decode.c`.

This function decodes single integer using the decoding handle `enc` and storing it into location `a`.

decode_longint()

```
int decode_longint ( Decode_handle *enc, long int *a );
```

Defined in `decode.c`.

This function decodes single long integer using the decoding handle `enc` and storing it into location `a`.

decode_doubles()

```
int decode_doubles ( Decode_handle *enc, double *array,
                     int n );
```

Defined in `decode.c`.

This function decodes a sequence of `n` doubles and stores them in `array`.

decode_double()

```
int decode_double ( Decode_handle *enc, double *a );
```

Defined in `decode.c`.

This function decodes a single double and stores it in `a`.

decode_end()

```
void decode_end ( Decode_handle *enc );
```

Defined in `decode.c`.

This function finishes the decoding process. Handle `enc` should not be used after the decoding process is finished.

encode_start()

```
void encode_start ( Encode_handle *enc, char *buffer,
                    int size );
```

Defined in `encode.c`.

This function initiates the encoding process. Parameter `size` should contain the size of the `buffer` in bytes and the address `buffer` should be evenly divisible by four.

encode_int()

```
int encode_int ( Decode_handle *enc, int a );
```

Defined in `encode.c`.

This function encodes a single integer and appends it to encoding stream `enc`.

encode_doubles()

```
int encode_doubles ( Decode_handle *enc, double *array,
                     int n );
```

Defined in `encode.c`.

This function encodes an array of doubles.

encode_double()

```
int encode_double ( Decode_handle *enc, double a );
```

Defined in `encode.c`.

This function encodes a single double, appending it to encoding stream `enc`.

encode_free()

```
size_t encode_free ( Encode_handle *enc )
```

Defined in `encode.c`.

This function returns the number of free bytes in the encoding stream `enc`.

encode_end()

```
void encode_end ( Encode_handle *enc );
```

Defined in `encode.c`.

This function ends the encoding process. Encoding stream `enc` should not be used after this function is called.

DEM_broadcast_MPI_event()

```
void DEM_broadcast_MPI_event ( MPI_Comm comm, int message,
                               int sock );
```

Defined in `mpi_events.c`.

This function broadcasts an MPI event to all the processors in the communicator `comm`. The MPI event is consisted of two integers, first is the unique identifier of the message and second is the file descriptor of the socket.

DEM_is_there_MPI_event()

```
int DEM_is_there_MPI_event ( MPI_Comm comm );
```

Defined in `mpi_events.c`.

This function return 1 if there are events present, otherwise the negative value of static variable `DEM_pause_flag` is returned, which signifies pause status. A Fortran wrapper for this function is provided.

DEM_collect_and_resend()

```
int DEM_collect_and_resend ( int major, int minor,
                             MPI_Comm comm, int sock );
```

Defined in `mpi_events.c`.

This function should be activated during the MPI event handling by the server processor. It gathers data from all the processors, which consists of an integer, and that many double numbers after the integer. The data is then prepended with the major/minor number pair, decoded using decoding routines and send to socket `sock`.

Function return 0 upon success.

DEM_MPI_event_handler()

```
int DEM_MPI_event_handler ( MPI_Comm comm, int rank );
```

Defined in `mpi_events.c`.

This function receives an MPI event message and acts correspondingly. It is a function which is really application-dependant, so in the future, it should be removed from the library and replaced with a general event-handler registration mechanism. A Fortran wrapper is provided for this function.

DEM_respond_client()

```
void DEM_respond_client ( int fd, void *buffer, int size );
```

Defined in `socket_events.c`.

This function sends data from the buffer to socket with file descriptor `fd`. Parameter `size` should specify the size of the buffer in bytes. The behaviour of this function is the following: Data is sent. If not all data is sent successfully, then `select()` is issued on the writing file descriptor with 10 sec timeout. If timeout occurs, the function returns without sending data. Otherwise, the rest of the data is being send, and until everything is transmitted, the procedure is repeated.

A return value for this function should be added in the future, so the server will know about all the data not being sent. This might trigger the attempt to resend pending data in later cycles. However, the DEM server, just abandons the data, if the network fails to retransmit any data within timeout period. This behaviour was adopted because it is the most robust behaviour as far as the numerical computation is concerned, and sending of data is regarded of much less importance, than continuous numerical run.

DEM_process_request()

```
void DEM_process_request ( struct DEM_socket *sock,
                           MPI_Comm comm );
```

Defined in `socket_events.c`.

This function again belongs to application specific section and should be replaced with a general handler-registration mechanism. It processes the socket generated events, and acts correspondingly. The communicator is used for issuing the MPI events if requested, and `sock` is the structure representing single client with all its attributes.

DEM_handle_client_request()

```
int DEM_handle_client_request ( struct DEM_socket *sock,
                                MPI_Comm comm );
```

Defined in `socket_events.c`.

This function is invoked if the signal for data is raised on certain socket. It reads the data from the socket and disconnects clients, which have disconnected themselves, or calls event processing functions if a regular client socket has been awaked. Return value upon success is 0, otherwise negative value is returned.

DEM_accept_new_client()

```
int DEM_accept_new_client(int sockfd);
```

Defined in `socket_events.c`.

This function is used for accepting new clients. Parameter `sockfd` is the listening socket to which the clients connect, and the returned value is the file descriptor of the new socket, which was created for the communication with this client. The socket is set to non-blocking mode and for the asynchronous communication, so the signal is raised when data is present.

DEM_event_handler()

`int DEM_event_handler (MPI_Comm comm);`

Defined in `socket_events.c`.

This is the main event handling routine. A `select()` system call is issued from here with timeout 0, because the signal was raised beforehand. The sockets which had some new data to read are then used in the calls to the further functions, which can either be the new client accepting routine or the `DEM_handle_client_request()` function is called.

A Fortran wrapper is provided for this function.

no_memory()

`void no_memory(void);`

Defined in `utilities.c`.

This is the error callback for the condition of insufficient memory.

DEM_is_there_event()

`int DEM_is_there_event (void);`

Defined in `utilities.c`.

This function returns non-zero value if the event on the socket is present, otherwise it returns zero. It is used in signal-driven event processing. A Fortran wrapper is provided.

DEM_atexit()

`void DEM_atexit (void);`

Defined in `utilities.c`.

This is the function which is registered as a clean-up function with the `atexit()` call. It cleans-up the listening socket.

DEM_io_signal_handler()

`void DEM_io_signal_handler (int signum)`

Defined in `utilities.c`.

This is the function, which is registered as a signal handler for the `SIGPOLL` signal. It sets the value of a global variable, which is returned by the `DEM_is_there_event()` function.

DEM_create_and_handle_socket()

```
int DEM_create_and_handle_socket ( MPI_Comm comm, int port,
                                   int mode );
```

Defined in `utilities.c`.

This function is used for the creation of the listening socket on the local `port`. MPI communicator is used for some additional buffer allocation for the buffered transmission operation and parameter `mode` is currently ignored and is left here for future extensions.

A Fortran wrapper is provided for this function.

C Input file format

C.1 DININIT.GEN

Rmin minimal radius of the particles, units are m,

Rmax maximal radius of the particles, units are m,

Len length of the particles, relative ratio,

dens density of particles (kg/m^3),

akn normal particle to particle contact stiffness,

bkn normal particle to wall contact stiffness,

rns ratio of shear to normal stiffness,

epsa coefficient of restitution for particle/particle contact,

epsb coefficient of restitution for particle/wall contact,

zs shear damping coefficient,

amn coefficient of friction for particle/particle contact,

bmn coefficient of friction for particle/wall contact,

dtfr fraction value of critical time interval,

gx acceleration in x direction (m/s^2), 0 for gravitation,

gy acceleration in y direction (m/s^2), 9.81 for gravitation,

Xmin the beginning of domain on x axis,

Xmax the end of domain on x axis,

Ymin the beginning of domain on y axis,

Ymax the end of domain on y axis,

strip length for spatial averaging,

tint time interval for time averaging.

C.2 DININIT1.ICS

This file is generated with utility program `Init`, which reads:

- total number of particles,
- minimum, maximum radius of the particles and number of types (sizes) of the particles (the latter shouldn't be 1 or Init crashes),
- maximum random ω,
- number of particles in the horizontal layer, (shouldn't be larger than total number of particles),

- coordinates of the particle in the left bottom corner,
- number of walls (15 or less in current implementation),
- coordinates of the walls (number of the wall, type, xw, yw, tw, h1, h2).[3]

The format of the actual `DININIT1.ICS` is the following:

nt Number of time intervals (the meaning is not perfectly clear),

nw number of walls in the simulation,

nw lines containing coordinates of the walls:

- number of the wall,
- `WType`, type of the wall (0 is the wall from which the particles bounce),
- `xw`, x coordinate of the point where the wall starts,
- `yw`, y coordinate of the point where the wall starts,
- `tw`, angle in degrees which shows direction of the wall (from the starting point in counterclockwise direction),
- `h1`, the thickness of the wall,
- `h2`, length of the wall from the starting point (xw,yw) in direction tw

NoTypes number of types of the particles,

NoTypes lines containing:

- number of the type,
- radius of particle of this type,

GloNoP total number of the particles,

GloNoP lines containing:

- number of the particle,
- the type of the particle,
- x coordinate of initial position of the particle,
- y coordinate of initial position of the particle,
- initial orientation of the particle (in radians),
- initial velocity of the particle in x direction,
- initial velocity of the particle in y direction,
- initial angular velocity of the particle ,

GloNoP blocks containing:

[3] Please not, that the walls must always face the particles in the right direction, as they would let particles go through, if they would approach the wall from the wrong direction. If it is observed, that the particles fall through the walls, the defining end-point of the wall should be set to the other end-point and the direction of the wall should be changed by 180 degrees.

- number of the particle and number of the neighbours of this particle (`NNb`),[4]
- NNb lines containing:
  - number of the neighbour,
  - the contact force in normal direction,
  - the contact force in surface direction,

C.3 DININIT.BCS

The format is the following[5]:

nttm number of relevant time intervals,

nttm blocks containing:

- `tr`, time interval
- `nw` (number of walls) lines containing:
  - velocity of the wall in x direction at this time interval,
  - velocity of the wall in y direction at this time interval,
  - angular velocity of the wall at this time interval,

ntts number of time averaging parameters,

ntts lines containing $ts(j)$, which are time averaging parameters,

nprt number of printout times,

nprt lines containing printout times.

C.4 DININIT.PAR

- first line: the net of processors,
- second line: scattering parameters,
- third line: two unknown parameters, set to (-1,-1).

C.5 DININIT.STRIP

This file contains the flag, the density, the four coordinates of the strip in which the stress is to be calculated and the critical pressure. The order is: x_1, y_1, x_2, y_2. The first number is the flag, and can be 1, which means that the calculation should be done, or 0 which disables the calculation. Density specifies after how time-steps, the stress should be calculated. In the last line is the critical pressure, on which the velocity changes[6].

[4]This can be initially set to zero.

[5]The format of this file must match the existing files as closely as possible, as it has been observed, that additional whitespace characters sometimes make the file unreadable by the program.

[6]This file has been tailored to the needs of the shear-box simulation.

C.6 DININIT.OUTFILES

This file contains the information about the output file generation. First line contains:

- flag, which is either 1 or 0 and specifies whether output file should be generated at all,
- density, which specifies after how many time-steps should the output file be generated,
- mode of the output file generation (1 is supported at the moment),
- the name of the global output file for the simulation.

The output file is written by the function `WriteOutputFile` which is defined in `io.F`, and other modes can be easily added there. A time-step number is appended to the name of the file.

Second line contains flag and the name of the stress output file. Flag being 0 means that no stress output file is generated, and flag value 1 means that output file should be generated.

The only type of the output file currently generated is the type 1, which has the following format:

- first line contains two integers, time-step and number of processors, `size`.
- `size` blocks which start with line containing the rank and the number of particles on that processor, and that many lines of (x,y) pairs of coordinates.
- `size` blocks which start with lines containing the rank and the number of particles on that processor, and that many lines of (u,v) pairs of coordinates.
- `size` blocks which start with lines containing the rank and the number of particles on that processor, and that many lines of omega coordinates.

In future, most probably the output files which would contain the value of the forces might be added.

D Programming notes

D.1 Socket communications

Reliable stream communication across sockets is used. Server listens to the socket on port 6666 by default, but this can be changed in the code. Several issues regard the buffer size and request/respond sizes. I tried to implement the protocol in such way, that requests and responds of any size are processed, therefore dynamic reallocation of buffers is necessary.

When the responds are sent, it is often, that they are of the size larger than the UNIX system socket buffer. Therefore, responds must be sent in more parts. The server behaviour is the following. It tries to send the whole buffer. If it doesn't succeeds, the socket is selected() for write condition, and when it is possible to write again, server repeats procedure with the remains of the respond. However, if select() doesn't selects the socket for writing within 10 seconds, the sending function fails, and computation goes on. This condition is rare however, since the client should be ready to read responds after it posts a request.

D.2 Signals

Signals are being used to probe for the events on the socket. The sockets are put into asynchronous mode upon creation, and when client has a request, the SIGPOLL signal is raised. This behaviour is (for now) different on the T3D, because I couldn't find the fcntl() or ioctl() request which would set this. Therefore, `DEM_is_there_event()` function is skipped on Cray, and `DEM_event_handler()` is called directly instead. There is only a minor overhead in this, since select() is called in non blocking mode afterwards, and function returns immediately if there is no request.

Other issue are the signal-critical sections of the program. I am aware, that some MPI implementation might have critical sections and signal handling at that point might not be a wise behaviour. If this is observer, the code for setting the sockets in asynchronous mode can always be taken out of the code, so the current T3D model can be used.

Other solution would be blocking the signal during MPI calls, but this is probably much more cumbersome.

As some of this issues are not even resolved according to the MPI standard, the user or future programmer should be aware of the possible problems that might arise due to signal use.

D.3 MPI issues

Buffered sends are used in the MPI communication and sufficiently large buffer has been attached to the MPI subsystem. However, the behaviour of the code, when the size of the buffer is too small for the communication, is undefined. Probably the functions would report a run-time error.

E Development notes for the new visualiser

visor_connect <name> <hostname> <port> If the call is successful, the connection is made to the specified port on the specified host. After the call, a new function called `<name>` is created.

visor_list Returns the list of all created connections.

If the call to `visor_connect` is successful, the connection is made to the specified port on the specified host. After the call, a new function called `<name>` is created with following subfunction:

name decode start/stop/int/double N Decodes data from the response buffer. Be sure, that the stream pointer is correctly moved to the beginning of the XDR encoded data. This can be assured by first calling `majmin` and the `2byte` subcommands, which is probably what a programmer would want anyway.

name disconnect Disconnects the created connection and destroys the Tcl object `name`.

name info Returns some information about the state of the connection. Mainly useful for debugging purposes.

name send Sends the major/minor pair. When data, not only the major/minor pairs are to be send to server, this functions needs additional capabilities.

name recv Reads data from socket and overwrites any previously read data.

name append Reads data from socket and appends it to the existing data in the receive buffer.

name majmin Returns the list with major and minor number of the received response. Moves stream pointer by two characters.

name 2byte Returns the 2byte value of the received response and moves the stream pointer.

name reset Resets the reading, erases the receive buffer and sets stream pointers to beginning.

name recv_size Returns the number of data in the receive buffer.

The other thing, the shared module implements is the *visorwidget*, which is the Tk widget for displaying DEM simulation. The new instance of the widget is created with:

```
% visorwidget pathName ?options?
```

The subcommands implemented beside the usual Tk widget commands are:

widget boundaries Expects four numbers to define the real coordinates which are boundaries for the simulation.

widget visor Links the widget with the visor passed as a parameter. This has to be performed at some point for the widget to be useful.

widget info Prints out some information about the state of the widget.

widget decode Initiates the decoding of data directly into the particle buffers of the widget.

widget npes Expects number, which is the number of processors in the simulation.

widget strip Expects four numbers which define the boundaries of the strip.

widget av_radius Expects a number, which represents the average radius of the particle.

widget walls Expects a list of lists made up of 5 elements which define the walls as described in appendix C.

The widget provides means of changing the colours and certain properties of the display via the usual Tk `-configure` switch.

F The Socklib Library

F.1 History

This functions were developed by Klaas Jan Wierenga in 1995 but no documentation was there. The author changed the Makefiles and split the sources, so it is a library now, not just one object file. During this process the documentation was written.

In addition, Fortran bindings were done, so this library can be called from Fortran as well, and used in communications between a Fortran and a C program over the network.[7]

F.2 How to call C functions from Fortran

Fortran passes everything by reference, which is the first fact, the wrappers must take into account. Fortran compiler on SGIs and Suns produces object code, with an underscore character appended to the name of procedure.

In order to call C functions in a library from Fortran, wrapper function should be created, with the same name as the C function with an additional underscore appended.

As far as parameters are concerned, this wrapper function gets all of them passed as pointers. If C function expects parameters by value, the wrapper function should pass on the values, not the pointers. If C expects parameter by reference, wrapper should just pass the addresses on as it receives them.

Return values from C are rather tricky and should be dealt with as an additional parameter. Therefore if a C function returns an integer which indicates status of the function operation, the wrapper function should have an additional parameter in which this result is stored. It is possible to get returned value directly back to Fortran as from a Fortran function call, but reportedly it is not portable, so with this library, an additional parameter model has been used.

Strings are passed to C as a non-null terminated strings, with length as defined in Fortran source. Extra characters are filled with whitespace characters. Wrapper function should expect this and deal with it. Another peculiarity is, that f77 compiler might append an additional parameter, which is the string length.

So current code uses strings without this extra information, and puts null-character at the point where first whitespace character occurs. With simple strings as hostnames, this should be sufficient.

Things are slightly different on the *Cray* platforms. Fortran compiler on the Cray doesn't generate the additional underscore, but changes the function name into all uppercase letters. Therefore an additional preprocessor macro has been defined: `__FORTRAN_DOES_UPPERCASE__`. If this macro is defined, then the symbols in the library, which represent the wrapper functions will not be the same as C functions with an underscore appended, but the C names transformed into uppercase.

Important consideration must also be given to the types. It seems that on most platforms, C tt double equals `double precision` in Fortran and C `float` equals Fortran `real`. How-

[7] Please note, that this library was not used in the project at all.

ever, on Cray the things are different again. Float in Fortran is equal to double in C and double precision in Fortran corresponds to long double in C.

F.3 C functions which are included in the libsock.a library

sock_client_connect()

```
int sock_client_connect(char *server_host, int server_port);
```

Function contacts server `server_host` on port `server_port`. The socket is created in SOCK_STREAM mode and on INET namespace. In case of error, -1 is returned, otherwise the returned value is the file-descriptor of the connected socket.

sock_server_listen()

```
int sock_server_listen(int *port);
```

This function creates listening socket in INET namespace, of SOCK_STREAM communication mode. The actual port which is used in communications is returned in `port`. Is might not be the same, if 0 is specified as a port number and arbitrary port is assigned. If valid port number is passed, it should be the one that's used. In case of error, -1 is returned, otherwise the file descriptor of listening socket is the return value.

sock_server_connect()

```
int sock_server_connect(int connect_sock, int block);
```

This function is used for accepting clients. If `block` is 1 then function stops and waits for a client to request connection on a listening port. If it is 0 it accepts the client if there is one, and return immediately, if there isn't one. Parameter `connect_sock` is the listening socket and `block` should be 1 or 0. The return value is -1 on error, or the file descriptor of the new socket. When connection is established, new socket is created, which will subsequently be used for communication, thus leaving the listening socket ready for any new incoming calls.

sock_close()

```
void sock_close(int sock);
```

Closes the socket `sock`.

sock_send()

```
int sock_send(int sock, const void *buf, unsigned nbytes);
```

Function `sock_send()` sends `nbytes` bytes from the buffer `buf` across socket `sock`, which should previously be connected. Function blocks until all data is transmitted. It return -1 on failure and number of transmitted bytes on success.

sock_recv()

```
int sock_recv(int sock, const void *buf, unsigned nbytes);
```

This function reads `nbytes` bytes into buffer `buf` from the socket `sock`. It blocks until all data is received. Return value is the number of bytes read, unless an error occurs and -1 is returned.

sock_xdr_send()

```
int sock_xdr_send(int sock, const void *buf,
                  int type, unsigned nitems);
```

This function writes data to socket and blocks until all data has been transmitted. Data is in the XDR format as defined by Sun RPC. Parameter `type` can be one of:

XDR_CHR or 0 Plain character

XDR_U_CHR or 1 Unsigned character

XDR_INT or 2 Signed integer

XDR_U_INT or 3 Unsigned integer

XDR_SHRT or 4 Short integer

XDR_U_SHRT or 5 Short unsigned integer

XDR_FLT or 6 Float

XDR_DBL or 7 Double

On error -1 is returned, otherwise the number of sent items. The named constants should be used with C, Fortran users can use the integer numbers, and pray to their Fortran gods, that they won't change.

sock_xdr_recv()

```
int sock_xdr_recv(int sock, const void *buf,
                  int type, unsigned nitems);
```

This function reads array of XDR elements from the socket and it blocks until all data has been received. The types allowed are the same as with `sock_xdr_send()`. On error -1 is returned, otherwise the number of received items.

sock_set_sndbuf()

```
int sock_set_sndbuf(int sock, int *sndsize);
```

This function changes the size of socket send buffers in the kernel. The size is specified in the address `sndsize`, and upon exit, the actual size of the new buffer, which may not be the same as specified is written there. Function returns -1 on error and 0 otherwise.

Larger buffer means more efficient transmission over fast networks.

sock_set_rcvbuf()

```
int sock_set_rcvbuf(int sock, int *rcvsize);
```

This function changes the size of socket receive buffers in the kernel. The size is specified in the address `rcvsize`, and upon exit, the actual size of the new buffer, which may not be the same as specified is written there. Function returns -1 on error and 0 otherwise.

_sock_set_options()

```
int _sock_set_options(int sock);
```

This is internal function and just sets default sizes for the receive and send buffers. No reason why should it be called from outside.

_sock_nonblock()

```
int _sock_nonblock(int sock);
```

Sets socket `sock` to non-blocking mode. Returns 0 on success and -1 on failure.

_sock_block()

```
int _sock_block(int sock);
```

Sets socket `sock` to blocking mode. Returns 0 on success and -1 on failure.

F.4 Fortran bindings to this library

All the functions from Fortran have an additional parameter `ERROR`, which will upon returning, contain the error number as it would have been returned from the C function.

The functions provided are:

```
CALL SOCK_CLIENT_CONNECT ( SERVERHOST, SERVERPORT, SOCK )
CALL SOCK_SERVER_LISTEN ( PORT, SOCK )
CALL SOCK_SERVER_CONNECT ( SOCK, BLOCK, NEWSOCK )
CALL SOCK_CLOSE ( SOCKET, ERROR )
CALL SOCK_RECV ( SOCKET, BUF, NBYTES, ERROR )
CALL SOCK_SEND ( SOCKET, BUF, NBYTES, ERROR )
CALL SOCK_XDR_SEND ( SOCKET, BUF, TYPE, NITEMS, ERROR )
CALL SOCK_XDR_RECV ( SOCKET, BUF, TYPE, NITEMS, ERROR )
CALL SOCK_SET_SNDBUF ( SOCKET, SNDSIZE, ERROR )
CALL SOCK_SET_RCVBUF ( SOCKET, RCVSIZE, ERROR )
```

```
CALL _SOCK_SET_OPTIONS ( SOCKET, ERROR )
CALL _SOCK_NONBLOCK ( SOCKET, ERROR )
CALL _SOCK_BLOCK ( SOCKET, ERROR )
```

F.5 Portability issues

This code has so far been compiled and tested on the following platforms:

- SUN SPARCstation with Sun OS 5.5.1,
- SUN station with Sun OS 4.1.3,
- SGI mips with IRIX 5.3,
- DEC alpha with OSF1,
- Meiko CS2 with Sun OS,
- Cray T3D.

References

[1] Mario Antonioletti. The dem application demonstrator. Technical report, EPCC, 1996.

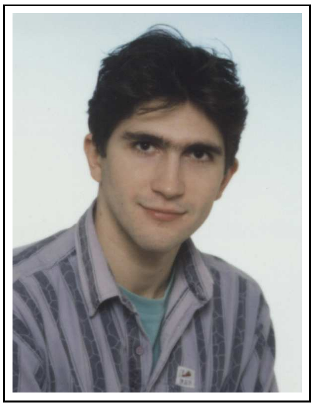
[2] Message Passing Interface Forum. Mpi: A message-passing interface standard. University of Tennessee, 12 June 1995.

[3] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[4] Graham Rong. *Discrete Element Modelling for flow of particulate materials in bins*. PhD thesis, University of Guelph, Canada, 1994.

[5] Thomas Schroder. Discrete Particle and Continuum Modelling of Particulate Solids. Technical Report SS-95-02, EPCC, 1995.

[6] W. Richard Stevens. *UNIX Network Programming*. Prentice Hall Software Series, 1990.



Timotej Ećimović will graduate in physics by the end of 1997 at the University of Ljubljana in Slovenia. His professional skills include various areas of computational physics, UNIX system administration and system programming, X Window programming, and many other common UNIX-platform script languages and utilities. His favourite platform is Linux.
When he's not working, Timotej enjoys aquarium fish, trips to nature, parties and guitar playing.

The supervisors of this project were Mr. Mario Antonioletti of the EPCC and Dr. Abdel Moniem Sanad of the Department of Civil Engineering at the University of Edinburgh. The author is very grateful for their help and support.