Interactive Computing for Engineering Applications

Jovana Knežević, Ralf-Peter Mundani
Technische Universität München
Lehrstuhl für Computation in Engineering, Arcisstraße 21, 80333 München
knezevic@bv.tum.de

Abstract: This study presents an integration framework for simulation codes that supports distributed computations as well as visualisation data access on demand in order to reduce latency and enable a high degree of interactivity based on a ‘minimal invasion’ principle, i.e. with only a few code changes necessary. Furthermore, we address the problem of rigid coupling / communication patterns of simulation back-ends with visualisation front-ends which, in case of huge data advent, leads, due to long response times, to a complete loss of the connection between cause and effect.

1 Introduction

Interactive computing refers to the real-time interaction of users with programs during program runtime in order to provide an immediate response to any kind of changes. Within the numerical simulation community, it is related to the practice of interactively exploring a computational experiment to gain insight concerning parameters, algorithmic behaviour, and optimisation potentials (Mulder, J. D., van Wijk, J. J., van Liere, R, 1999). This usually happens at the interface between mathematics, computer science, and engineering and involves good knowledge especially of numerics, algorithms and data structures, high-performance computing, and visualisation.

Tools such as CSE (Van Liere, R., van Wijk, J. J., 1996), SCIRun (Parker et al., 1999; Abrams et al., 2007), CUMULVS (Kohl, J. A., Wilde, T., Bernholdt, D. E., 2006), or RealityGrid (Brooke et al., 2003) allow users to integrate their simulation codes for an interactive steering without the need to be an expert in all aforementioned domains. Commonly agreed central requirements are to support development and execution, thus, to design two parts separately responsible for each requirement. The development (i.e. front-end) part consists of sophisticated user interface which guides the user in building a solution to his problem, while on the back-end, high-performance computers are running often time- and memory-consuming simulation tools (Figure 1).

1.1 Related work

CSE (Van Liere, R., van Wijk, J. J., 1996) is based on the idea that the data manager provides a subscribe / notify interface to inform all the components of changes made in the data and an interactive graphics editing tool allowing users to sketch an interface and bind variables to user interface components.
SCIRun (Parker et al., 1999; Abrams et al., 2007) is a problem solving environment for modelling, simulation, and visualisation of scientific problems. The user has to set up a network of modules and can interact via their corresponding graphical user interface. Several parameters can be changed during the simulation without the need to stop it. Other changes with a deeper impact on the simulation require an automatic cancellation and restart of the simulation.

CUMULVS (Kohl, J. A., Wilde, T., Bernholdt, D. E., 2006) is a middle layer between the application program and the visualisation and steering front-end. It encompasses all the connection and data protocols needed to dynamically attach multiple visualisation and steering front-ends to a running application. Its basic principle is to have the user declare in the application which parameters are allowed to be modified or steered during the computation.

In the RealityGrid project (Brooke et al., 2003) an application is structured into the client, the simulation and the visualisation module communicating by means of a steering library. It is designed to simplify the changes necessary to make an existing code “computationally steerable”. Attempts to overcome the problem of rigid coupling/communication patterns of simulation back-ends with visualisation front-ends have resulted in the insertion of check- and break-points (Brooke et al., 2003, e.g.) at fixed places in the code where modified parameters are fetched and the simulation has to be restarted, respectively.

Figure 1: At the development, front-end, the user guides the simulation in building a solution to his problem, while on the back-end, a high-performance computer (HPC) is running time-consuming iterative program.

Despite of the number and the assortment of the existing libraries, problem solving environments and application frameworks, these tools are limited in their possible applications and mostly entail heavy code changes in order to integrate an existing program.
2 The idea of the platform

In order to leverage interactive computing for a broader scope of applications, the immediate response of the simulation side to the changes made by the user is indispensable. Therefore, if coupled to our platform, the regular course of the C/C++ simulation is being interrupted in small, cyclic intervals ensued by a check for updates. If, meanwhile, there has been no user activity, the control is given back to the computation, which continues from the previous interrupt-point either until the stage when the results should be sent to the user process, or until the expiration of another time interval.

On the other hand, if any user interaction has taken place, the new data is received. The receipt of the message is considered to be instantaneous. What is required as the next step is the update of the old data. However, it is the responsibility of the user himself to instruct the simulation program, before compiling and running it, to match the received data to the simulation-specific requisites. Afterwards, the whole computation is intended to be restarted by manipulating the iteration vector \( i = (i_{d1}, i_{d2}, \ldots i_{dn}) \), i.e. the loop indices \( i_{d} \) of all loops by setting it for each loop index to the predetermined maximal value, as shown in the following pseudo code:

\[
\begin{align*}
\text{for } (t \leftarrow T_0 \text{ to } T_N) \text{ do} & \quad \text{// iterations over time} \\
\text{for } (i_{d1} \leftarrow X_{10} \text{ to } X_{1N}) \text{ do} & \quad \text{// alarm is set beforehand to some interval} \\
\text{for } (i_{d2} \leftarrow X_{20} \text{ to } X_{2N}) & \quad \text{process(data[\[i_{d1}\][ i_{d2}]] \quad //can be interrupted at any point, } i_{d} \text{ are reset in the interrupt handler}
\end{align*}
\]

Nevertheless, when the control of the execution is given back to the main computation, it is obliged to continue at the point where it has previously been interrupted. Conveniently, this happens only until the end of current, most-inner loop iteration, where the earliest opportunity to compare the current value of the loop index with maximum value and analogously exit all the loops (i.e. starting with most-inner one and finishing with the most-outer one) is exploited. In further steps of the algorithm (i.e. a new iteration), all the loop indexes are reinitialised with zero or some other simulation-specific initial value, thus, the computation is resumed. Nevertheless, to guarantee the correct execution of the program, one has to be aware of at least two important matters when handling interrupts.

The first important issue is to ensure that certain types of objects which are being modified both in the signal handler and the main computation are updated in an uninterruptable way. Namely, such updates are said to be atomic, thus, it is impossible for the object to be in an inconsistent state during the update. However, the types that support atomic updates are usually very simple (e.g., integer). The C standard provides the specific type for this purpose. Although this topic gets into operating system dependant problems, no matter how the operating system handles the issue, using this certain type for loop counters one can be sure he will not end up with corrupted bytes due to interrupts.

The other issue is that with C/C++ compilers if a loop is testing the same memory address repeatedly, it would almost certainly arrange to reference memory once only, and copy the value into a hardware register, thus speeding up the loop. If the value in
the signal handler is changed, the old value in the register is used again instead of reading the updated value from the memory, which results in undesired behaviour of our program. To advantage, the usage of a specific type qualifier is supported by the C standard. It will tell the compiler that the object is subject to sudden change for reasons which cannot be predicted from a study of the program itself, and will force every reference to such an object to be a genuine reference.

When one or several iterations have been finished without an interrupt, the new results are copied into the send buffer and forwarded to the user process. One more time it is the user’s responsibility to instruct the program, this time the one running at the front-end, how to interpret the received data correctly so that it can be visualised.

The following section describes the simulation patterns following either shared memory (e.g. OpenMP/POSIX threads) or “hybrid” parallelisation models (i.e. MPI and OpenMP), where the aforementioned algorithm leads to synchronisation problems among the single threads of control and results in an extra necessity of the “restart”-signals being sent from the thread which is first informed about the changes and has received the updates to the rest of the threads.

2.1 Multithreading and “hybrid”-parallelisation scenarios

Many applications that are amenable to concurrent execution can be programmed using either shared memory or message passing algorithms.

In the former case, Lee (2006) stated that “although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability and determinism. Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that nondeterminism.”

In our integration framework, with OpenMP / POSIX threads, e.g., the idea is that as soon as a random thread is interrupted at the expiration of the user-specified interval (Figure 2 a), it checks, using the functionality of the Message Passing Interface, if some information regarding the user activity is available. If the aforesaid probing of the user’s message indicates that a change has been made, the receiving thread instantly obtains information about it. Furthermore, all the other threads are notified respectively that their computations should be started anew.

Similarly to the case of a single thread of execution, as soon as an interrupted thread becomes aware that there is a user-made change which has to be applied, it automatically sets its loop counters in such way that the computation will be restarted immediately after the end of the current most inner-loop step. Concerning OpenMP parallel index-loop, our implementation, favourably, issues its clean termination and clean exit from the parallel region immediately after the threads are either implicitly or explicitly synchronised and before the whole iteration is resumed. In this manner, the correct program execution will be ensured.
When computing environments supporting both parallel paradigms mentioned at the beginning of this subsection are available, a hybrid algorithm, at a sufficiently high level of granularity, may be used to advantage (V. S. Sunderam, 1990). It can be observed that nowadays computing environments which possess the hardware diversity required for such parallel applications, and also provide a pillar for multiple concurrent computation models are rather the norm than an exception.

In the case of “hybrid” parallelisation of a simulation, a random thread in each active process is being interrupted (Figure 2 b), hence, fetches an opportunity to check for the updates. If no user action has taken place, its previous computation is continued without stopping the other threads. Otherwise, the user’s data is received, the signals for the restart of all the computations are sent and its private loop counters are manipulated in the way that it is assured that in the next step its own computation is taken up equally.

The difference in comparison to the multithreaded parallelisation is that now all the processes have to be notified about the user action, which involves additional message passing overheads. In the current state of development, it is assumed that one master process, which is the direct interface of the user’s process to the computing-nodes (i.e. slaves), apprises all the slave processes of the user-made alterations, which, in the case of demand for and availability of a large number of slaves may result in the master process becoming a bottleneck.

![Diagram](image)

Figure 2: On front-end the identifiers of the user interaction are kept in a queue and sent regularly to the computing node(s). a) A random thread is being interrupted in regular intervals and has an opportunity to react on change; b) One of the processes on the back-end receives the change first and informs the others; all the computing processes analogously react on change and synchronise at the end of each iteration.

Therefore, an efficient broadcast algorithm for transferring the signal to all computing nodes has to be developed. Moreover, in efforts to interrupt one thread per process, an inevitable trade-off between ensuring a minimal number of checks per process and allowing for the opportunity to receive the data promptly has to be faced.
3 **A cellular automaton test case**

To test the above concepts, we have coupled our platform with a simple C++ cellular automaton simulation and its visualisation using the Qt library. The initial settings, which include only the dimensions of the grid, the position and the size of the inlet and the outlet, are specified by the user as command line input parameters.

3.1 **Initial settings**

The simulation itself runs as a separate process and consists of the iterations which run over time – in each, all the cells being updated sequentially according to the information from the previous iteration as well as a random number of new cells being added.

At the beginning, the process on the user’s side is ready to visualise only the inlet and the outlet, however, as soon as it receives the simulation data, i.e. state of all cells, it visualises this information as well (Figure 3 a). To elaborate more on this, the attributes of the inlet and the outlet objects are stored in the memory of both front-end and the back-end process, thus, it is prepared for the visualisation immediately. On the other hand, the cell-data, which is only stored on the simulation side, is copied appropriately to the send buffer and sent to the front-end after a user-defined number of iterations. As already pointed out in the previous section, it is the user’s process responsibility to interpret the received message and accurately visualise it, thus, the user is expected to adequately instruct the program according to his specific demands.

3.2 **User interaction**

As described in the Section 2, the user has an opportunity to interact with the simulation via front-end graphical user interface. For the cellular automaton simulation, the actions have been restricted to adding and moving obstacles via mouse interaction (Figure 3 b and 3 d), as well as moving inlet and outlet via a keyboard (Figure 3 c).

While the user interaction is in progress, the information about it is being stored in a FIFO (First In, First Out) structure, letting the structure items be sent serially to the simulation process and thereupon removed from the storage.

On the other side, the simulation process is being interrupted in regular, 1 millisecond-intervals in order to check if any update is waiting be received. However, for the stability reasons, what is considered atomic and cannot be interrupted is processing of a single cell. Therefore, in the interrupt handler it has to be ensured that the processing of the update takes place exclusively in between treatments of the two successive cells. This property is achieved by repeatedly trying to obtain a lock resource and then giving momentarily the control back to the computation, until once the resource is successfully acquired. At this point, a check for an update can be performed, the change received, and the loop counter manipulated so that in the next step the current
iteration is resumed.

![Figure 3: User interaction in the Cellular Automaton simulation – a) an initial scenario (inlet in the upper left, outlet in the bottom right), b) adding blocks of obstacles by mouse-press, c) moving inlet and outlet by the keyboard left/right/up/down arrow, d) moving the obstacles by mouse-move.](image)

As previously mentioned, when the control is given back to the cells’ processing routine, as soon as the one with the current index is finished, the loop index is reinitialised with zero, leading to the instant resumption of the iteration, involving the new settings. Nonetheless, if there has been no user activity, the saved state is restored and the current computation proceeds.

Once the user-defined number of iterations have been finished without the interference, the results are copied into a send buffer and sent to the user process in order to be interpreted and visualised.

### 3.3 Code modification requirements

To integrate the platform into any application scenario, a few modifications of the code have to be made by the user. Since these modifications are only minor, we list all of them. First of all, all the loop indexes which will be affected by the interrupt handler have to be declared global and both atomicity of their updates and prevention of the compiler optimisations which would lead to incorrect value references insured. Second
of all, the integrity of each user-defined ‘atomic’ sequence of instructions in the simulation code have to be ensured. Furthermore, the calls to our send and receive functions have to be included in the appropriate places in the programs. However, the user himself has to instruct the interpretation of the data (e.g. in the receive buffers). Finally, he has to enable the regular checks for updates by including appropriate functions which will examine and change the default signal (interrupt) action and specify the time interval in which these checks should be made.

4 Conclusion

In this paper, we have presented a generic platform which couples engineering simulation and visualisation tools in the way which allows user to trigger the simulation during the execution and receive immediate feedback with only a few code changes necessary. Although this is work in progress, results for the first, sequential, test case look very promising. Nevertheless, the question of the signal transfer from the user’s to all the computing nodes will be part of upcoming research, as well as the integration and testing of the framework incorporated into several parallel engineering simulation scenarios.

5 Acknowledgements

This work has been financially supported by Munich Centre of Advanced Computing (MAC) and the International Graduate School of Science and Engineering (IGSSE) at Technische Universität München.

6 References


