Distributed multiscale computing

Borgdorff, J.

B

Technical details of the MUSCLE 2 runtime environment

To increase the separation between the model and the runtime environment each mapper or submodel has its own instance controller that will do the actual communication with other parts of the simulation. When an instance controller starts up it first tries to register to the Simulation Manager. It then queries the Local Manager for the location of all the instances that it has a sending conduit to. The Local Manager will then query the Simulation Manager in a separate thread if it does not know the location. When an instance is finished, its instance controller will deregister it at the Simulation Manager.

Although each instance controller and thus each instance uses a separate thread by default, it is also possible to implement submodels asynchronously. MUSCLE 2 will be able to manage a large number of light asynchronous submodels in a small number of threads. This leads to both lower memory usage and faster computation since there are far fewer thread context switches but it makes the submodel code slightly more complex and, if not properly coded, prone to race conditions.

Error handling, throughout the program, is designed to work fail-fast. If an uncaught exception occurs in one instance, MUSCLE 2 assumes that continuing the simulation will not yield valid results and it will try to shut down all other instances. This behaviour was implemented to prevent wasting resources on systems that charges

end users for the total wall-clock time used by a simulation. It also prevents deadlocks when an instance still expects data from another that has already quit. MUSCLE 2 does not provide error recovery, instead each submodel should handle its own checkpointing, if needed.

B.1 Implementation of the MUSCLE Transport Overlay (MTO)

The MUSCLE 2 Transport Overlay (MTO) is a C++ user space daemon. It listens for connections from MUSCLE 2 on a single cluster, and keeps in contact with MTO's on other clusters. It forwards any data from MUSCLE 2 intended for another cluster to that clusters MTO. To identify the MTO associated to a MUSCLE 2 TCP/IP address, each MTO mandates a separate port range to MUSCLE 2.

The default connection between MTO's uses plain non-blocking TCP/IP sockets, and this is well tested. To optimise speed over wide area networks, it has a local buffer of 3 MB and it will prefer sending over receiving up to the point that it will not allow more incoming data if the send buffers are too large or numerous. The MPWide 1.8 [59] library is optionally enabled for connections between MTO's. MPWide is a library to optimise message-passing performance over wide-area networks, especially for larger messages. This option currently only works between a pair of MTO's and the performance depends on the connection between the clusters, but there are ongoing efforts to increase the compatibility.

B.2 QosCosGrid and MUSCLE 2 integration

We identified two main integration points of the QosCosGrid software stack and MUSCLE 2. First, the location (IP address and port) of the MUSCLE Simulation Manager can be exchanged automatically with other MUSCLE Local Managers via the QCG-Coordinator service - a global registry that offers blocking call semantics. Moreover, this relaxes the requirement that the Simulation Manager and Local Managers must be started in some particular order. The second benefit of using the QosCosGrid stack with MUSCLE is that it automates the process of submission of cross-cluster simulations by: co-allocating resources and submitting on multiple sites (if available, using the Advance Reservation mechanism); staging in- and output files to and from

every system involved in a simulation; and finally, allowing users to peek at the output of every submodel from a single location.

B.3 Comparison between MUSCLE 1 and MUSCLE 2

The largest changes in MUSCLE since MUSCLE 1 involve decoupling functionalities. The separation between the library and runtime environment makes the system more usable, since users now do not need to go through MUSCLE internals to do basic operations like getting model parameters, and this in turn makes submodel code less susceptible to being incompatible with newer versions of MUSCLE. The separation of C/C++/Fortran code from the main Java code makes compilation much more portable. Finally, the separation of message passing code and the communication method allows choosing more efficient serialisation and communication methods when able.

In terms of portability, MUSCLE 2 comes with all Java prerequisites so they do not have to be installed manually. Moreover, the number of required Java libraries has been drastically reduced. Notably, MUSCLE 2 no longer relies on the Java Agent Development Environment (JADE) for its communication. This way, the MUSCLE 2 initialisation sequence and communication routines are more transparent, which in turn lead to numerous performance enhancements to communication protocols and serialisation algorithms. As a result, MUSCLE 2 can handle messages up to a gigabyte, while MUSCLE 1 will not handle messages larger than 10 MB. Although distributed execution was already possible with MUSCLE 1, it only worked for specifically set up environments, whereas MUSCLE 2 will run with most standard environments.

In MUSCLE 1 the Java Native Interface (JNI) was used to couple native instances. Although JNI is an efficient way to transfer data from and to Java, it gave MUSCLE 1 usability and portability issues and introduced incompatibilities with OpenMP and MPI. In MUSCLE 2, submodels must link to the MUSCLE 2 library instead, at a penalty of doing communications between Java and C++ with the somewhat slower TCP/IP.

Additional new features of MUSCLE 2 include a CMake-based build system, having standardised and archived I/O handling, more flexible coupling, and automated regression tests.