Comparison of 32bit vs 64bit integer MUMPS in Optistruct

Kostas Sikelis

Altair, Greece

Abstract

In the world of simulations there is a constant need for accurate results. In order to meet this challenge engineers resort to finer models, which put the solving abilities of commercial software to the test. Almost all well established and widely accepted commercial solvers today, were conceived and designed many years ago, at an era when state-of-the-art computing resources were at their infancy. Consequently in many cases they carry limitations which reflect their long past. Since then, hardware has evolved dramatically and simulation software is struggling to keep up.

One of the issues which early software engineers failed to address consistently or even take into account, is the size of the integer variable type. The default value has always been 4 bytes, which limits the value of an integer variable to approximately 2 billion (or 4 billion for the unsigned case). This limit sufficed for decades, but it seems the trend is shifting. In fact more and more cases are constantly popping up in the user community, where 4 byte integer overflow is causing simulation runs to be aborted.

Apparently, there are two solutions to the problem. The first one is to identify all places where overflowing integers are used and promote them to 8 bytes. In a solver consisting of millions of lines of code, written by numerous developers at different periods of time, this is easier said than done. At least, it involves many developers, to go through the code and make appropriate changes. Unless done carefully, this approach can lead to disaster, consuming huge amounts from the teams resources in terms of developing and testing. The obvious advantage is that the new executable will be able to optimally handle the previously failing jobs, by properly utilizing more memory only where it may be necessary. Moreover, although initially the modified code can be developed as a separate executable, once it is stable enough, it can fully replace the previous one, without imposing extra burden on the QA system.

A second alternative is to take a brute force approach and promote all integers of the code to 8 bytes. This approach carries significant advantages. For the Fortran parts there is a simple compiler option which automatically performs the conversion. For the C/C++ files a trivial script can convert all integers to long integers. Of course the above two steps can be assigned to one person who can work on the new "promoted" executable leaving the remaining team undistracted. However, the solution is not without drawbacks. Modern software interface 3rd party libraries which are not always available in long integer arithmetic. Therefore the interface points have to be identified and taken care of so that inter-library communication is consistent. Furthermore, the new executable would be suboptimal since it will always allocate 8 bytes for the integers which in most cases will be redundant.

Nevertheless, the simple and clean design of the second approach has caused vendors to adopt it, so nowadays many commercial software (and 3rd party libraries) come in i8 flavor. Recently this approach was taken in Optistruct to pre-actively address 4 byte integer limits and the new executable was made available to public with 2017.0 release.

In our quest, one of the external libraries that had to be interfaced was MUMPS, which has been selected as the default direct solver used in OS. In fact the 64bit capabilities of MUMPS solver have been utilized for quite some time in the SMP version of OS, thus removing the overflow problems from inside the MUMPS solver. However, the MPI version of OS strictly remained 32 bit, since no precompiled version of MPI was available, preventing a clean porting of MPI applications to 64 bits. This barrier was relieved by Intel in their MPI implementation where the addition of the ilp64 keyword in the running command, automatically forces all integers in MPI functions to be treated as long integers.

This capability is offered only for Fortran based MPI applications, but fortunately MUMPS is implemented in Fortran. Compiling MUMPS for the ilp64 flavor of MPI was pretty straight forward, with the exception of a couple of limitations, posed by the MPI system itself (presented in Appendix A) and which quite logically had not been anticipated by the MUMPS team. However, due to MUMPS's superb design it was trivial to identify and fix the problematic points.

In the first section of the presentation we show a comparison of the 32bit integer vs the 64bit integer versions of MUMPS using both SMP and MPI executables and discuss the extra cost involved in the 64bit integer arithmetic. We argue that it is not entirely unreasonable to eventually drop the default i4 executable and relieve our QA system from the extra burden.

In addition we will present some tests of the new BLR factorization capability of MUMPS. Indeed this a promising development in MUMPS with the potential to address huge industrial models with less resources and more speed. We discuss our experience.

Appendix A

Intel MPI supports 64-bit integers in the fortran version of MPI. The 64-bit libraries can be loaded by adding keyword -ilp64 in the mpirun command.

Currently the following limitations exist which affect MUMPS source code:

1. In REDUCE operations MPI_2INTEGER data type is not promoted, therefore the send and receive buffers must be INTEGER(4)

2. For custom reduction functions used in REDUCE operations the integer arguments must be INTEGER(4) For more details refer to the intel mpi reference guide.

The above limitations cause the following changes in MUMPS source code. These apply to MUMPS version 5.01

i)	bcast_errors.F@16 :	INTEGER IN(2), OUT(2) ⟶ INTEGER(4) IN(2), OUT(2)
ii)	tools_common.F@269 :	INTEGER TEMP1(2), TEMP2(2) ⟶ INTEGER(4) TEMP1(2), TEMP2(2)
iii)	[dzsc]fac_scalings_simScale_util.F@23 :	INTEGER IWRK(IWSZ) ⟶ INTEGER(4) IWRK(IWSZ)
	[dzsc]fac_scalings_simScale_util.F@433-436:	INTEGER LEN, INV(2*LEN), INOUTV(2*LEN), DTYPE ⟶ INTEGER(4) LEN, INV(2*LEN), INOUTV(2*LEN), DTYPE
	[dzsc]fac_scalings_simScale_util.F@466 :	INTEGER IW(IWSZ) ⟶ INTEGER(4) IW(IWSZ)
	[dzsc]fac_scalings_simScale_util.F@923 :	INTEGER IWRK(IWSZ) ⟶ INTEGER(4) IWRK(IWSZ)
iv)	[dzsc]mumps_driver.F@1642 :	INTEGER TMP1(2),TMP(2) ⟶ INTEGER(4) TMP1(2),TMP(2)