FAQ: Compiler Problems

This section covers the Intel, PGI and GNU compilers as pointed out for each section.

Why does my IA32 executable fail when static arrays are very large?

Note: This is becoming (mostly) obsolete upon migrating to 64 bit.

When having more than around 1 GByte of static arrays, one starts to obtain segmentation faults when starting up the program. On IA32, it should however be possible to use data until the process memory limit of 2 GByte is reached. This presently is only a problem with the g95 compiler on 32 bit systems, but some alternatives for other compilers remain documented below.

Workaround:

1. **Use static linkage:** This is the recommended workaround. The static linkage flag is -Bstatic for the PGI compiler, and -static for the GNU and Intel compilers.
2. **Use dynamic allocation:** Replace
   
   ```fortran
   DOUBLE PRECISION A(150000000)
   by
   DOUBLE PRECISION, ALLOCATABLE :: A(:)
   INTEGER :: ISTAT
   ...
   ALLOCATE(A(150000000), STAT=ISTAT)
   IF (ISTAT /= 0) THEN
   STOP 'ALLOCATION FAILURE'
   END IF
   ...
   DEALLOCATE(A, STAT=ISTAT)
   IF (ISTAT /= 0) THEN
   STOP 'DEALLOCATION FAILURE'
   END IF
   ...
   ``

   This is the right way to do things in Fortran 90, at least for newly developed code; you can now adjust A to the size really needed.

3. **Intel Compiler:** Add a common statement to your array declaration as shown here

   ```fortran
   DOUBLE PRECISION A(150000000)
   COMMON /COM_INTEL/ A
   ```

   (more than one array may of course appear in the same common block) and specify the -Qdyncom"COM_INTEL" option to the Intel Compiler. Please note that this will work together with the -g debug option. This is a known bug.

4. **pgf77 Compiler:** Add a dynamic common statement and allocate that

   ```fortran
   DOUBLE PRECISION A(150000000)
   COMMON, ALLOCATABLE /COM_PGI/ A
   ...
   ALLOCATE(/COM_PGI/, STAT=ISTAT)
   IF (ISTAT .NE. 0) THEN
   STOP 'ALLOCATION FAILURE'
   END IF
   ...
   This uses a Fortran extension available only for pgf77, not for pgf90.

5. **g77 Compiler:** A compilation problem is fixed since the 3.1 release of the gcc.

Code fails to link ("Relocation truncated to fit")

This may happen on x86_64 based systems if your data segment becomes larger than 2 GBytes. For the Intel compiler, please use the compiler options `--mcmodel=medium --shared-intel` to build and link your code. The `-fpic` option should be avoided. Other compilers (GCC, PGI) should simply use `--mcmodel=medium`. For NAG Fortran, try `-Wc,-mcmodel=medium`. Note that this problem does not arise if you manage memory on the heap, so we recommend converting static arrays to allocatable ones.

Using large temporary arrays in subroutines fails

For the Intel compiler, using large automatic arrays as in

```fortran
subroutine foo(n, u)
  integer :: n
  real(rk) :: u(n)
  ...
  real(rk) :: temp(n)
  ...
end subroutine
```

leads to segmentation faults and/or signal 11 crashes of the generated executables. The reason is that automatic arrays are placed on the stack, and the stack limit may be too low.
Workarounds:

1. Use the `-heap-arrays` compiler switch to move allocation to the heap. You can also specify a size modifier if only large arrays should be thusly treated, i.e. `-heap-arrays 10000` would place arrays larger than 10000 Bytes on the heap.
2. Increase the stack limit via the command `ulimit -s unlimited`. Note that special measures might be needed for MPI parallel programs to propagate this setting across nodes.
3. Change over to use dynamic allocation:

   ```fortran
   subroutine foo(n, u)
   integer :: n
   real(rk) :: u(n)
   ...
   real(rk), allocatable :: temp(:, )
   ...
   allocate(temp(n), ...) ! allocation status query omitted here, please check to be safe
   ...
   deallocate(temp)
   end subroutine
   ```

   this will use the heap for the required storage.

**icc and icpc fail to compile my (assembler) code**

icc and icpc do their best to behave like the GNU compilers. However, they do not support assembler. If this causes trouble, use the `-no-gcc` compiler switch. This will disable the gcc macros, and hence suppress using assembler statements which are (usually) shielded by macro invocations.

**My program stops in I/O when writing or reading large files**

Your file may be larger than 2 Gbytes and hence beyond the 32 bits supported by the traditional open() system call. Linux nowadays does support file sizes larger than 2 GBytes, however you may need to recompile your program to use this feature.

1. **GNU C compiler (gcc)**: Please recompile all sources containing I/O calls using the preprocessor macro `_FILE_OFFSET_BITS=64`, i.e.

   ```bash
   gcc -c -D_FILE_OFFSET_BITS=64 (... other options) foo.c
   ```

   See [this page](#) for further details (some of which may be outdated).

2. **PGI Fortran compiler**: Please use the `-Mlfs` compiler switch when linking.

3. **Intel Fortran compiler**: Automatically supports large files. However, there are limits on the record sizes.

4. **On 64 bit systems** in 64 bit mode no problems should occur since large files should be supported by default. Note however that there still may be limits for accessing large files via NFS.

**I’ve got a lot of Fortran-unformatted Files from big-endian systems (old vector or IBM Power). Can I use those?**

Yes. There are two variants of this situation:

1. **Portability of unformatted data**. In this situation you want to use both Intel (little endian) and other (big endian) platforms concurrently.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGI</td>
<td>Use the compilation switch <code>-Mbyteswapio</code></td>
</tr>
<tr>
<td>Intel</td>
<td>Set the following environment variable (under sh, ksh, bash) <em>before</em> running your executable: <code>export F_UFMTENDIAN=&quot;big&quot;</code></td>
</tr>
<tr>
<td>GCC</td>
<td>Set the following environment variable (under sh, ksh, bash) <em>before</em> running your executable: <code>export GFORTRAN_CONVERT_UNIT=&quot;big_endian&quot;</code></td>
</tr>
</tbody>
</table>

   In this case all unformatted files are operated on in big endian mode.

2. **Migration from one platform to the other**. Here you need to write a program to convert your data from (or to) big endian once and for all. In the following we shall assume that conversion happens from big endian to little endian, and unit 22 is used to read in the big endian unformatted data.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGI</td>
<td>Use the OPEN statement specifier <code>CONVERT</code> in your source: <code>OPEN(22, FILE='mysundata', FORM='UNFORMATTED', CONVERT='BIG_ENDIAN')</code></td>
</tr>
<tr>
<td>Intel</td>
<td>Set the following environment variable (under sh, ksh, bash) <em>before</em> running your executable: <code>export F_UFMTENDIAN=&quot;little:big:22&quot;</code> <em>This will switch I/O to big endian on unit 22 only.</em></td>
</tr>
<tr>
<td>GCC</td>
<td>Set the following environment variable (under sh, ksh, bash) <em>before</em> running your executable: <code>export GFORTRAN_CONVERT_UNIT=&quot;native:big_endian:22&quot;</code> <em>This will switch I/O to big endian on unit 22, and use the native endianness on all others.</em></td>
</tr>
</tbody>
</table>
Please note that you need to perform testing on data files from more exotic big endian platforms because assumptions still are made on IEEE conformance and Fortran record layout.

Generally the Intel and GCC Compilers gives you more flexible handling since the functionality is supported by the run time environment and no code recompile is required. You can also specify more than one unit via a list of comma-separated values, or a range of units, i.e. 10-20. Note that the conversion procedure has a significant impact on I/O performance.

Please also refer to a section below on how to use this functionality in conjunction with MPI.

Reading unformatted direct access files generated on other HPC platforms

While the above mentioned method works fine for unformatted sequential files, care must be taken to read unformatted direct-access files generated on other platforms. When a direct access file is opened, the parameter: ...access='DIRECT', recl=irecl,... is required, specifying the record length. The unit irecl refers to is implementation dependent; E.g., 4 Byte words on Itanium using the Intel compiler, 8 Byte words on the VPP. It is therefore good practice to set the parameter irecl before the open call via the inquire function. Assume the largest record one wants to write is an array A, which was declared as:

real, dimension(n):: rval

Then one should add the following line before the open call:

inquire(iolength=irecl) rval

and use irecl in the following open statement. Thus the assigned record length for the direct-access file becomes independent of the implementation.

Maximum record length for unformatted direct access I/O for Intel ifort

Up to compiler release 10.1, Intel's documentation does not provide any information on this. The maximum value is 2 GBytes ($2^{31}$ bytes) for each record; note that the storage unit used is 4 bytes unless the switch -assume byterecl is specified, in which case the storage unit is 1 byte.

Compiler does not optimize as specified

The Intel compiler may occasionally give the complaint "fortcom: Warning: Optimization suppressed due to excessive resource requirements; contact Intel Premier Support". In this case, please try the -override-limits switch. However, this may lead to very long compilation time and/or considerable memory usage. If system resources are overstrained, the compilation may fail anyway. If compilation completes, the generated code may be incorrect. In the latter two cases please send your source file(s) to the LRZ support team.

Gradual underflow optimization: -ftz compiler option may improve performance

This applies to usage of the Intel compilers; the material is drawn from the SGI document "Linux Application Tuning Guide", Chapter 2, The SGI Compiling Environment. Many processors do not handle denormalized arithmetic (for gradual underflow) in hardware. The support of gradual underflow is implementation-dependent. Use the -ftz option with the Intel compilers to force the flushing of denormalized results to zero.

Note that frequent gradual underflow arithmetic in a program causes the program to run very slowly, consuming large amounts of system time (this can be determined with the 'time' command). In this case, it is best to trace the source of the underflows and fix the code. Gradual underflow is often a source of reduced accuracy anyway.

When starting my binary, it complains about missing libpgc.so (IA32/Opteron/EM64T)

This is a PGI compiler issue and hence only relevant for x86 and possibly x86_64 systems. Please load the environment module fortran/pgi/x.y, or ccomp /pgi/x.y using the version number x.y you used for compiling the application. This should correctly set the LD_LIBRARY_PATH variable.

Intel C, C++ or Fortran compilers: Linkage fails

This not uncommonly happens if you need to link against system libraries (e.g., libX11, libpthread, ...). Of course there are many possible reasons:

1. Check whether you have specified all needed libraries
2. Check whether you are trying to link 32 bit objects into a 64 bit executable. This is not possible.
3. If you use the -static option of the compiler in your linkage command, please remove it or replace it by -static-intel to only link the Intel libraries statically.

See also the linkage problems with MPI below for further information.

When starting my binary, it complains about missing symbol (e.g., 0_memcopyA)

This can be a problem when using non-default versions of the Intel Compilers, or mixing different versions of the C and Fortran compilers. When doing e.g.,

module switch intel/-non-default version>

for compilation, this setting must also be performed before execution of the program. Otherwise the wrong base library may be bound at run time; in fact if the order of library entries in $LD_LIBRARY_PATH$ is wrong it may happen that the wrong library is bound from the C installation for a Fortran program (or vice versa). There are a number of possibilities to deal with this problem.

1. ifort supports the -static-intel link time switch which statically links in the Intel libraries. However, static memory is then limited on a 64 bit system.
2. Use the -Xlinker -path [path_to_libraries] switch at linkage to fix the path chosen for resolution of the shared libraries. We're considering to make this the default setting in the compiler configuration file.
How to get an error traceback for your code

If you are using Intel version 8.1 (and higher) compilers, the `-traceback` option should get you a traceback if your code fails. Adding the `-g` option may provide source line information as well. You can also add `-fpe0` if you suspect that your code fails due to floating point exception error. Note that all of the above can (and perhaps should) be specified in addition to any options used for the production code. Example:

```
program sample
 real a(10), b(10)
 do i=1,10
   b(i)=0.0
   a(i)=a(i)/b(i)
 end do
 stop
end
```

```
$ ifort -fpe0 -traceback -g sample.f $ ulimit -c unlimited $ a.out

fortrnl: error (65): floating invalid
Image PC Routine Line Source
 a.out 4000000000002D11 MAIN__ 6 sample.f90
 a.out 4000000000002A80 Unknown Unknown Unknown
 libc.so.6.1 2000000000435C50 Unknown Unknown Unknown
 a.out 40000000000027C0 Unknown Unknown Unknown
```

Note that the ulimit -c setting is necessary if you want to investigate a core dump.