No. 191
December 2015

Bits & Bytes

Max Planck Computing & Data Facility (MPCDF)*
Gießenbachstraße 2, D-85748 Garching bei München

High-Performance Computing

Ingeborg Weidl, Florian Merz (Lenovo), Markus Rampp

Formatted I/O in Fortran

In general, using formatted I/O for large amounts of data (e. g. big arrays) should be avoided, because the files get bigger and it is much slower compared to unformatted I/O. In certain situations, however (e. g. when running a legacy code or for debugging), it can still be necessary or useful to write formatted output. The purpose of this article is to discuss I/O performance tuning possibilities for those situations.

In Fortran, every new line that is written to a file corresponds to a record that has to be written to the file system. If a lot of (short) lines are created, e. g. by writing every element of a large array to a file in an implicit loop, this large number of small write requests can become a significant performance bottleneck. To speed up the I/O, the number of write requests has to be reduced. Apart from changing the output pattern (e. g. by reducing the number of lines that have to be written by putting more data in every line), this can be achieved without source code modifications by enabling I/O buffering in the Fortran runtime environment. I/O buffering aggregates many small records before an actual write request to the file system is issued, so that the effective number of write requests to the file system is reduced. Note that this means that most recent data will not appear in the file system immediately, but all data is guaranteed to be flushed into the file when the file is closed. The default behaviour of the Intel Fortran compiler is to do no buffering. We have recently introduced an environment variable in the Intel Compiler modules on *hydra* that switches on I/O buffering for Fortran (FORT_BUFFERED=yes). For further Intel runtime options for Fortran I/O, see the respective FORT_ environment variables on the Intel web page. I/O buffering is switched on by default for gfortran.

Buffer Aliasing in MPI calls

According to the message-passing interface (MPI) standard arguments of MPI functions that are modified (tagged OUT or INOUT in terms of the MPI standard) must not be 'aliased' with any other argument, i. e. the memory locations of the corresponding buffers may not overlap. The MPI-2 standard introduced the tag MPI_INPLACE, which allows usage of the same buffer (variable) for input and output buffers (e. g.). Nevertheless, quite a number of applications apparently still use buffer aliasing without specifying MPI_INPLACE which used to be silently tolerated by many MPI implementations. However, modern MPI implementations such as the version 1.4 of the IBM Parallel Environment which was introduced as the default on the HPC system *hydra* earlier this year, or recent releases of Intel MPI, finally enforce standard conformance in this respect, which leads to crashing of non-conforming codes. Code developers are strongly advised to adhere to the MPI standard and to correct non-conforming MPI calls. As a fallback for such applications which cannot be modified, version 1.3 of the parallel environment (non-default module mpi.ibm/1.3) is still available on *hydra*.

How to archive/backup data on the HPC cluster hydra

On *hydra*, there are two global, parallel file systems of type GPFS (/u and /ptmp) and a migrating GPFS file system /r which should be used to archive or backup your data in /u and /ptmp. As there is no regular system backup of /u and no system backup of /ptmp, the users are responsible to do their own backups.

The migrating file system /r (a symbolic link to /ghi/r) is available only on the login nodes *hydra.rzg.mpg.de* and on the interactive nodes *hydra-i.rzg.mpg.de*. Each user has a subdirectory /r/⟨initial⟩/⟨userid⟩ to store his/her data. For efficiency, smaller files should be packed to tar, cpio or zip files (with a size of at least 1 GByte up to 500 GBytes) before archiving them in /r. When the file system /r gets full above a certain value, files will be automatically transferred from disk to tape, beginning with the largest files which have been unused the longest time. There is a limit of 120,000 files in /r per user.

*Tel.: +49(89) 3299-01, e-mail: benutzerberatung@mpcdf.mpg.de, URL: http://www.mpcdf.mpg.de/
Editor: Renate Dohmen

If you access a file which has been migrated to tape, the file will automatically be transferred back from tape to disk. This of course implies a delay. You can manually force the recall of a migrated file by using any command which opens the file. You can recall in advance all files needed by some job with a command like

```
file myfiles/*
```

You can see which files are resident on disk and which ones have been migrated to tape with the command `ghi_ls` (located in /usr/local/bin), optionally with the option `-l`. Here is a sample output:

```
hydra01% ghi_ls -l
G -rw-r--r--    1   ifw        rzs                22 Nov 21 15:12 a1
H -rw-------    1   ifw        rzs       138958551040 Sep 18 22:22 abc.tar
H -rw-r--r--    1   ifw        rzs        1073741312 May 06 2009  core
G -rw-r--r--    1   ifw        rzs                 0 Jun 20 2008  dsmerror.log
B -rw-r--r--    1   ifw        rzs        1079040000 Aug 03 2010  dummyz3
```

The first column states where the file resides: a 'G' means the file is resident on disk; an 'H' means the file has been transferred to the underlying HPSS tape archiving system; a 'B' means premigrated to tape (the file has already been copied to HPSS, but is still present on disk, however may be removed by the system if disk space is needed).

Please note: If you want to 'tar' files that are already located in /r, please carefully check in the contents of the resulting TAR file whether all migrated files were correctly retrieved and included into the TAR file. Don't use 'gzip' or 'compress' on files that are already located in /r. It's not necessary, because all files are automatically compressed by hardware as soon as they are written to tape.

See also: http://www.mpcdf.mpg.de/services/computing/hydra/filesystems

The MPCDF GitLab Service

Thomas Zastrow, Florian Kaiser

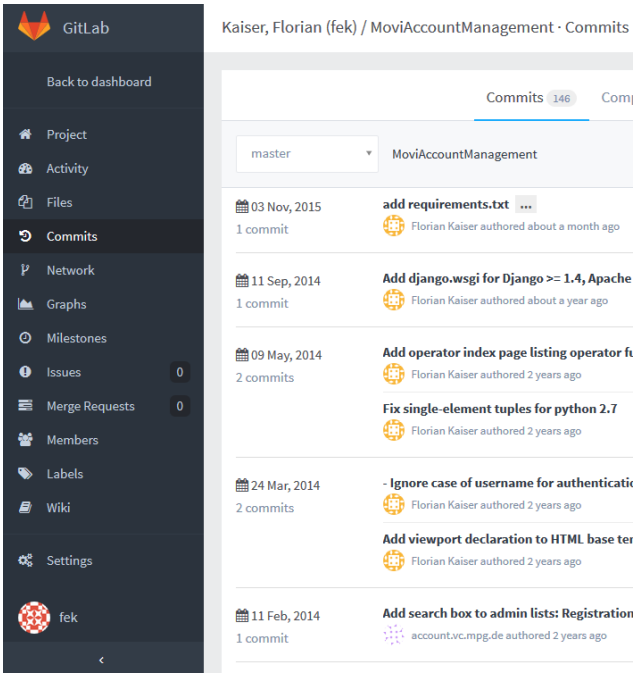

Figure 1: GitLab navigation bar

Since November this year the MPCDF offers a new Git-based version control service on the basis of the Open-Source software GitLab. GitLab offers a convenient web interface for managing and controlling software projects, similar to the popular GitHub service. However, the service is hosted entirely on-premise at the MPCDF, with the user in full control over projects and other information. Existing Git-based projects can be imported into GitLab.

An 'Activity tracker' summarizes recent activities on a project. Enhanced visualization and management functions are supporting common workflows of a version control system. Additionally, GitLab offers a built-in wiki as well as an issue tracker. An extensive user and group management enables team work and collaboration. Figure 1 shows the navigation bar of GitLab and the functionality it offers.

The GitLab service can be used by any MPCDF user. At the moment, however, only user accounts that have MPCDF's DataShare service enabled are allowed to log into GitLab. This is a temporary workaround until we have reworked our subscription service. Allowing external guest users is not yet enabled, but is planned for the near future. A Continuous Integration System will be added to the GitLab installation soon. For further information, please contact us (support@mpcdf.de).

Quick Links

- MPCDF's GitLab service
- GitLab documentation
- MPCDF's subscription service (subscribe to the DataShare service for the time being to enable GitLab)

Upgrade of the Visualization Infrastructure

Klaus Reuter, Ingeborg Weidl

Starting from January 2016, interactive GPU-accelerated visualization services will be offered on a subset of the *hydra* supercomputer. In particular, a number of the *hydra* GPU nodes were upgraded with additional main memory and with local hard disks for temporary user data. Compared to the visualization cluster *VIZ*, users will experience increased performance due to more recent GPUs (2 x NVIDIA Tesla K20X per node, 6 GB GPU RAM each) and more powerful host systems (2 x 10-core Intel IvyBridge CPUs per node, up to 256 GB RAM). In addition, the integration of the visualization services into the supercomputer enables innovative approaches such as in-situ visualization.

Interactive visualization sessions are to be requested using a special batch queue on the system, similar to compute jobs. A sample submit script will be made availabe on the MPCDF web page. As soon as the requested visualization session starts on *hydra*, the user will receive a notification E-mail with instructions. Similar to *VIZ*, VNC is the method of choice to connect remotely to the VNC desktop running on *hydra*. Popular visualization software will be provided. The reservation calendar used on *VIZ* won't be available on *hydra*.

Finally, after more than five years of successful operation, the visualization cluster *VIZ* is scheduled to be decommissioned by the end of January 2016. Users are encouraged to migrate their data from viz:/u and viz:/vizdata to the corresponding *hydra* file systems. The data gateway to IFERC – '*viztrans*' – will remain in operation with a slightly modified configuration, though. Details will be announced separately.

New Centre of Excellence: NOMAD Laboratory

Raphael Ritz, Hermann Lederer

On October 27th and 28th, 2015, the kick-off meeting of the new European Centre of Excellence called **NOMAD Laboratory** took place at the Harnack-Haus of the Max Planck Society in Berlin-Dahlem.

Lead by Matthias Scheffler, director at the Fritz-Haber Institute of the Max Planck Society, eight top-level research facilities (including two Max Planck Institutes) and four high-performance computing centres (amoung them MPCDF and LRZ) have joined forces to accelerate the search for new materials suitable for new technological developments and materials science research in general.

The Centre of Excellence will develop methods to systematically and efficiently screen and analyze the huge amounts of data that have been generated in materials science – and that continue to grow with increasing rate currently. This will make it possible to discover currently unknown materials with interesting properties as well as new phenomena in known materials. Results of these activities will be fed into a virtual encyclopedia of materials which is one of the ultimate goals of the Centre of Excellence.

MPCDF's main contributions will be in the areas of visualization and high-performance computing infrastructure. Here, the MPCDF can build on long-standing expertise in these fields in general as well as on its experience gained in supporting the **NoMaD Repository** over the past year. This repository provides the data on which the NOMAD Laboratory will operate. The MPCDF will also continue to support the further development of this repository in close contact and coordination with the new Centre of Excellence.

Quick Links:

- NOMAD Laboratory website
- NoMaD Repository website
- MPG news feature announcing the Centre of Excellence
- Interview with Matthias Scheffler, PI of the Centre of Excellence (in German)

Mailing problems with new DSL

Andreas Schott

Several DSL providers are delivering pre-configured routers or are updating the configuration of these routers when a contract has changed, e. g. from DSL to VDSL. Some of these configuration parameters can cause trouble within the MPG setup by for example blocking E-mail traffic to the MPG E-mail servers. If your mail is (suddenly) no longer working at home, please check the configuration of your DSL modem. There may be entries to setup a firewall blocking access to E-mail related ports for encrypted pop, imap or smtp (995, 993, 465) or even to use a list of trusted E-mail servers, on which the MPG servers are not listed. In case you encounter such problems and the above hints are not sufficient, please contact our helpdesk indicating that there was a contract or hardware exchange of your DSL.

Debugging Memory Corruption with the Address Sanitizer Library

Lorenz Hüdepohl

Since GCC 4.8 programs can be instrumented to use the 'AddressSanitizer', a fast memory-corruption detector. It detects out-of-bound accesses to heap, stack and global memory regions as well as invalid usage of any memory that has already been freed again.

This is especially valuable for C programs, where the language does not offer built-in support to check for out-of-bounds accesses. Consider for example the following invalid C program 'test.c':

```
#include <stdio.h>
int main(int argc, char *argv[argc]) {
    int i;
    char a[12];
    for (i = 0; i <= 12; i++) {
        a[i] = 0;
        fprintf(stderr, '%d ', i);
    }
    return 0;
}
```

The error here is that the condition i $\langle =$ 12 in the for-loop permits `i` to be equal to 12 which results in a write access to the non-existing array element `a[12]` (the correct condition would of course be i $\langle$ 12).

The invalid write now sets the byte behind the last element of a to zero. Whether this memory location is writeable or not – the latter would be good, as this then would trigger a segfault, an unambiguous sign that something is wrong with your code – or whatever other meaning the program assigns to this memory location is architecture and compiler specific.

In this example it can even happen (amd64/gcc/no optimization) that `&a[12]` is exactly the address of the loop variable `i`. The surprising result is that now the loop never terminates, as the statement `a[12]= 0` in the last iteration zeros `i` again:

```
$> gcc -g test.c -o normal
$> ./normal
0 1 2 3 4 5 6 7 8 9 10 11 0 1 2 3 4 5 6 7 ...
[ ... loops forever ... ]
```

This is exactly the kind of error that can be detected with the address sanitizer. Compile the program with `-fsanitize=address` to see its effect:

```
$> gcc -g -fsanitize=address test.c -o asan
$> ./asan
0 1 2 3 4 5 6 7 8 9 10 11 =================================================================
==28624==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffee084820c at
                                  pc 0x000000400ac9 bp 0x7ffee08481b0 sp 0x7ffee08481a8
WRITE of size 1 at 0x7ffee084820c thread T0
    #0 0x400ac8 in main test.c:6
    #1 0x7f3eef42a78f in __libc_start_main (/lib64/libc.so.6+0x2078f)
    #2 0x400928 in _start (asan+0x400928)

Address 0x7ffee084820c is located in stack of thread T0 at offset 44 in frame
    #0 0x400a05 in main test.c:2

  This frame has 1 object(s):
    [32, 44) 'a' <== Memory access at offset 44 overflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind
      mechanism or swapcontext
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow test.c:6 main
Shadow bytes around the buggy address:
[ ... further output omitted for brevity ... ]
```

ASan has detected the invalid access, warns the user and aborts the program. Note that in this particular example, other memory debugging tools might not report any errors, as only the program's own memory (`a[]` and `i`) is accessed.

Valgrind, for example, relies on interpreting (+ JIT compiling) the program's machine code, it can thus not distinguish the invalid write to `i` via `a[12]= 0` from a valid `i = 0`. For this, instrumentation of the program during compilation is necessary.

Valgrind in particular is valuable for other tasks, though, for example to detect the use of uninitialized memory and memory leaks, profiling and pthread synchronization errors. Also, at least a subset of out-of-bounds accesses (to invalid or non-writeable memory regions) are also detected. For more information about Valgrind, see also the previous article 'Code validation tools' from Bits & Bytes issue 183.

Note that there are also some out-of-bound conditions that cannot be detected by ASan. The tool works by inserting guard bytes around any distinct memory buffer. This cannot be done in-between adjacent structure members, in order to preserve binary compatibility of the structure's memory layout. An out-of-bound access within a `struct` is therefore not detected!

In Fortran, the language has some built-in notion of the length of arrays which enables comparatively robust out-of-bounds checking via special compiler switches (e. g. `-fcheck=bounds` for GCC). However, if you by mistake explicitly lie about the bounds to the compiler, also this cannot save you. Consider the following Fortran example, where in subroutine `testsub` both bounds of array `a` are explicitly specified:

```
module m1
  implicit none
  contains

  subroutine testsub(a, n)
    integer :: n, i
    integer :: a(1:n)
    do i = 1, n
      a(i) = 0
    end do
  end subroutine

end module

program test
  use m1
  use iso_fortran_env, only : error_unit
  implicit none
  integer :: b(4), a(4)

  a(:) = 1.0
  b(:) = 2.0

  call testsub(b, 6)

  write(error_unit,*) a
  write(error_unit,*) b

end program
```

The example is invalid, as in the main program we lie to `testsub` about the bounds of `a`, resulting again in out-of-bounds writes. Note that this is also not caught by the built-in array bounds checks offered by the compiler:

```
$> gfortran -g -fcheck=all test.F90 -o normal
$> ./normal
  0  0  1  1
  0  0  0  0
```

From the output of the program it can be seen that the array `b` now has been erroneously written to by `testsub`, as `b` was placed adjacent in memory to `a` in the particular compiler/architecture combination used here.

With the address sanitizer these kinds of errors can be detected:

```
$> gfortran -g -fcheck=all -fsanitize=address test.F90 -o asan
$> ./asan
=================================================================
==30474==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fff0ab65b60 at
                                  pc 0x000000400dd8 bp 0x7fff0ab65890 sp 0x7fff0ab65888
WRITE of size 4 at 0x7fff0ab65b60 thread T0
    #0 0x400dd7 in __m1_MOD_testsub test.F90:9
    #1 0x400fd5 in test test.F90:24
    #2 0x4011d0 in main test.F90:16
    #3 0x7fc4e0eee78f in __libc_start_main (/lib64/libc.so.6+0x2078f)
    #4 0x400ba8 in _start (asan+0x400ba8)

Address 0x7fff0ab65b60 is located in stack of thread T0 at offset 112 in frame
    #0 0x400e3a in test test.F90:15

  This frame has 2 object(s):
    [32, 48) 'a'
    [96, 112) 'b' <== Memory access at offset 112 overflows this variable
```

```
HINT: this may be a false positive if your program uses some custom stack unwind
      mechanism or swapcontext
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow test.F90:9 __m1_MOD_testsub
Shadow bytes around the buggy address:
[ ... further output omitted for brevity ... ]
```

Again, Valgrind would not be able to detect a fault, as only valid memory locations were accessed:

```
$> valgrind ./normal
==30480== Memcheck, a memory error detector
==30480== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==30480== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==30480== Command: ./normal
==30480==
           0           0           1           1
           0           0           0           0
==30480==
==30480== HEAP SUMMARY:
==30480==     in use at exit: 0 bytes in 0 blocks
==30480==   total heap usage: 26 allocs, 26 frees, 29,017 bytes allocated
==30480==
==30480== All heap blocks were freed -- no leaks are possible
==30480==
==30480== For counts of detected and suppressed errors, rerun with: -v
==30480== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

We suggest never to use the idiom where both array bounds are specified as dummy-arguments to a subroutine (specifying only the lower bound is fine). Instead, use so-called assumed-shape arrays which are available since Fortran 90:

```
subroutine testsub(a)
     integer :: a(:)
     integer :: i
     do i = 1, size(a)
       a(i) = ...
```

Or with a lower-bound possibly different from 1 by:

```
subroutine testsub(a, lb)
     integer :: i, lb
     integer :: a(lb:)
     ...
     do i = lb, ubound(a, dim=1)
       a(i) = ...
```