No. 199
August 2018

Bits & Bytes

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

High-performance Computing

Ingeborg Weidl, Hermann Lederer

HPC Clusters Cobra and Hydra

The new HPC system Cobra, based on Intel Skylake processors, was put into production in April 2018. In May, the cluster was extended by another 636 'Skylake' nodes in a fifth island with full non-blocking fat-tree OmniPath interconnect. In total, there are now 127,520 cores with a total main memory of close to 0.5 PB and a peak performance of close to 10 PetaFlop/s. Six login nodes provide access to the system. The next step in this system's expansion will be the addition of nodes with NVIDIA Volta GPUs. These nodes are expected to be available before the end of the year and will act as a replacement for the 5 years old Kepler K20X GPUs in Hydra.

For the Hydra system a large part of the Intel IvyBridge based nodes have been de-commissioned during the last months in order to make room for Cobra. The reduced Hydra system now consists of 338 nodes with 676 K20X GPUs, and 272 general compute nodes. Queue waiting times on Hydra have therefore increased noticeably.

The evolution of the Anonymous FTP server and data sharing at MPCDF

John Alan Kennedy

After many years in service the anonymous FTP server at the MPCDF (ftp.rzg.mpg.de) has been upgraded to a modern host system. Although the server remains available, there have been some slight modifications to the system and users are also advised to use the MPCDF Datashare service in future to solve data sharing use cases.

The major modifications to the ftp service are:

1. The FTP server is now restricted to pure anonymous mode.
2. Users with MPCDF accounts are now required to use the more secure SFTP protocol.

This means that anonymous users can still access the FTP server to anonymously upload/download data, but users who wish to access with an MPCDF account, and thus control which data may be shared, are now required to use the more secure SFTP protocol.

The FTP server will remain in service and provides a tried and tested means of sharing data with anonymous external users. However, as we mentioned earlier, the new Datashare service can also be used to cover the core use cases, and we will highlight these below.

The two major use cases are:

1. Sharing data with Anonymous external users (outgoing data)
2. Data upload from Anonymous external users (incoming data)

The MPCDF FTP server solves these use cases by allowing anonymous access to the server for data upload, but dictating that data for download must be managed by MPCDF users, which now use the SFTP protocol.

These use cases can equally be covered by the MPCDF Datashare service which now allows password protected links to be created for anonymous upload (dropbox like function) and sharing. The benefit of using the Datashare system is that it allows more granularity w.r.t. data management. A user can create different links for different datasets and share them with different external collaborators.

For more information see the following links for Anonymous FTP and Datashare

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

Lightweight performance measurement tools

Lorenz Hüdepohl, Sebastian Ohlmann

In the following we shall describe two lightweight profiling tools, likwid (developed by RRZE) and ftimings (developed by the MPCDF) which facilitate performance analysis and guide optimization of application codes on the level of compute node(s). Likwid is a mature and well-known tool suite which has been widely used in the community over many years. With the recent introduction of a new backend for accessing hardware counters, likwid has become more generally usable and can now be supported on MPCDF HPC systems. Our in-house developed ftimings library takes a slightly more minimalistic approach. In fact, it is very similar in spirit to the venerable perflib developed by the RZG/MPCDF in that it provides a simple API allowing the user to instrument sections of interest in the code and to measure timings and related information for these sections.

1) The likwid tool suite

Likwid is a lightweight, open-source performance monitoring and benchmarking suite, developed for the x86 architecture at the regional computing center Erlangen (RRZE). The source code is available at https://github.com/RRZE-HPC/likwid. Its focus lies in measuring performance on single nodes; not on communication patterns in distributed-memory systems (i. e., not on MPI performance). Important use cases for likwid are getting information on the memory hierarchy of a node, measuring hardware performance counters, and pinning processes and threads to certain cores.

Information on the memory hierarchy of a node, i. e., on its topology, can be obtained with

```
likwid-topology -g
```

which gives information on the CPU type, the number of sockets, cores, and threads, as well as sizes and groups of different cache levels and NUMA domains, including a graphical view. This information can then be used, e. g., to design the layout of arrays or loops for optimal cache usage.

Hardware performance counters can be measured using the `likwid-perfctr` tool. For example, the memory bandwidth that an application utilizes can be obtained with

```
likwid-perfctr -g MEM ./a.out
```

which will print out a summary of measured counters and derived quantities at the end of the run. Different groups of counters are predefined to measure, e. g., the number of floating-point operations (FLOPS_DP/FLOPS_SP for double/single precision), the memory bandwidth (MEM), the arithmetic or operational intensity (number of floating point operations per transferred byte of memory; MEM_DP/MEM_SP for double/single precision), transfer rates between different cache levels (CACHES), the branch prediction miss rate (BRANCH), or the load-to-store ratio (DATA). All available counter groups can be displayed with `likwid-perfctr -a`. The information obtained by measuring these characteristics can be used to analyze the performance of the application and find potential bottlenecks. Moreover, these global measurements can be performed without adapting the code. Measurements of the hardware counters can be restricted to certain code regions by instrumenting the code using the marker API, available for C and Fortran 90. The marker API is activated by passing the option `-m` to `likwid-perfctr` (for more information, see the online documentation).

Pinning of threads to specific cores can be useful for multi-threaded applications to avoid context switches and to have defined behaviour with respect to the NUMA layout of the node. This is offered by likwid-pin as a unified interface to applications using pthreads or OpenMP with gcc or intel compilers. Pinning can be done on several levels of the NUMA hierarchy (`likwid-pin -p` gives more information) and to physical and/or logical cores (when using Hyperthreading). To pin all threads of an OpenMP-parallel application to the 40 physical cores of a Broadwell node on Draco, one can use

```
likwid-pin -c N:0-39 ./a.out
```

Pinning can be combined with measuring performance counters by specifying the corresponding expression (e. g., N:0-39) as an argument with `-C` (capital letter) to `likwid-perfctr`. The likwid tool suite also provides `likwid-mpirun` as a tool to use pinning and measure performance counters for MPI and hybrid MPI/OpenMP applications. It is considered experimental according to the documentation, but has been tested for various combinations of schedulers, compilers, and MPI libraries (e. g., SLURM + Intel compiler + Intel MPI). To measure floating point operations per second for a pure MPI application, one could use

```
likwid-mpirun -np 80 -g FLOPS_DP ./a.out
```

to run on two Draco Broadwell nodes (this replaces the srun call). This will print out information on the performance counters for each core and a summary for the total run. If the application is instrumented using the

marker API to track code regions, the switch `-m` is recognized here as well. More information on options for each of the commands and more functionality of likwid can be found online at https://github.com/RRZE-HPC/likwid/wiki.

Usage on MPCDF systems

The likwid tool suite is available on the MPCDF HPC systems Draco and Cobra and can be accessed as a module (on Cobra only after a compiler module has been loaded). It is compiled to use the Linux tool 'perf' as a backend for measuring performance counters. After issuing `module load likwid`, the binaries are available in `$LIKWID_HOME/bin`, the library files for the marker API in `$LIKWID_HOME/lib`, the include files in `$LIKWID_HOME/include`. To avoid interference with other users, measurements should always be done on full nodes. If you want to measure the number of floating point operations per second, you should use nodes with the Broadwell architecture or newer (e. g., Cobra or Broadwell partition on Draco), because these counters are not reliable on earlier Intel architectures (Sandybridge, Ivybridge, Haswell).

2) ftimings, a simple Fortran library for time measurement and profiling

Very often applications need some simple and cheap timing information, both for internal purposes, such as cleanly shutting down (checkpointing) before batch system time limits are encountered, or for profiling the various parts of the code. For this, we developed a simple library at the MPCDF called `ftimings`, that is primarily aimed at Fortran-based codes. It is similar in spirit to the venerable `perflib` in that it uses explicit calls placed by the user in the code to instrument sections of interest. It always measures wall-clock time, and calls from within OpenMP parallel regions are only considered for the first thread.

The `ftimings` library provides a simple object-oriented Fortran interface, and can also be used from a C code or from C code sections of a Fortran program. A specific feature of `ftimings` is that timing sections can be nested and will be presented consistently as a tree in the output.

Example. A very minimal mock-up of a code could be instrumented like this:

```
program phi
  use ftimings
  type(timer_t) :: timer
  [...]

  call timer%enable()

  call timer%start('init')
  call some_init()
  call timer%stop('init')

  call timer%start('main-loop')
  do i = 1, 4
    call timer%start('a')
    call a()
    call timer%stop('a')

    call timer%start('b')
    call b()
    call timer%stop('b')

    call c()
  end do
  call timer%stop('main-loop')

  call timer%print()
end program
```

The `call timer%print()` statement could produce an output such as this:

```
loh@cobra01:~> ./a.out
 /= Group                                          [s]     fraction
 |                                          ============ ============
 |_ [Root] (running)                          18.002003        1.000
   |_ (own)                                      0.000250        0.000
   |_ init                                       2.000139        0.111
   |_ main-loop                                 16.001608        0.889
     |_ (own)                                     4.000575        0.250
     |_ a  (4x)                                   4.000535        0.250
     |_ b  (4x)                                   8.000498        0.500
```

As you can see, the information is stored in a tree, and nested `timer%start()` calls appear as child nodes of their encompassing sections. This, of course, necessitates that the user does not close an outer section before all started inner sections have been closed. The user is responsible for ensuring that this is the case. However, the library detects such misuse and disables the affected `timer_t` object after printing a warning message.

Every node is labeled with the argument given to the `%start()`/`%stop()` subroutines and by default lists two values, its time duration and the fraction this particular duration is relative to its parent node. It is also possible to record various other resource data, such as allocated memory or FLOP counts (on hardware that supports it).

The time values for multiple identical `%start()`/`%stop()` pairs in the same encompassing section are accumulated together and increase a counter value, visible in parenthesis after the node names `a` and `b`. The mean time duration for a single call of the `b` subroutine would thus be about 2 seconds.

The measurements are constructed such that all child nodes on the same level exactly add up to the value of their parent node, sections of the code that are not enclosed by timing sections are put into a special node

(own). Here, this represents the loop overhead and the time spent in the c subroutine.

Run-time querying. In addition to this profiling functionality, the library also allows access these values from within the program while it is running. This can, for example, be used to schedule the stop of the program after a certain duration, or to abort it should a certain operation take an abnormal amount of time.

For this, the procedure %since() is available, that returns the amount of seconds since an unclosed %start() call has been made with the supplied label. Example:

```
print *, 'It is ', timer%since('main-loop'), &
  ' seconds ago since start(''main-loop'') ' &
  'has been called'
```

There is also the %get() method to query already closed sections:

```
call timer%start('init')
call some_init()
call timer%stop('init')

print *, 'some_init() took', &
  timer%get('init'), ' seconds'
```

Hierarchical queries. To query child nodes, additional arguments with their labels should be supplied. For example, to get the time spent in all those b calls, use

```
print *, 'All the b() calls took', &
  timer%get('main-loop', 'b'), ' seconds'
```

(The same scheme can be used for the %since() method)

Additionally, there are functions to sum up all the time spent in any descendant nodes with a given name. Suppose you enclosed all communication parts of your code with %start/%stop('comm') calls, you can query the total amount of time spent in those with the %in_entries() method:

```
print *, 'Spent', &
  timer%in_entries('comm'), &
  ' seconds in communication'
```

In order to consider only those sections below a certain parent, additional arguments should be provided before the label name to be queried,

```
print *, 'Spent', &
 timer%in_entries('main-loop', 'b', 'comm'), &
 ' seconds in comm sections in b()'
```

Sorting. By default, arguments are inserted into the tree in the order in which their %start() calls were done. To get a better overview in very large trees covering a whole simulation code, it is often better to sort the resulting tree levels. For this, there is the %sort() method that sorts the internal tree structure. Note that the original order is then lost.

Thresholds. Additionally, sometimes one is not overly concerned with many sections that cover only a short time duration. It is possible to exclude those by passing the optional argument threshold to the %print() method of timer_t. Then, all child sections that took less than that threshold are subsumed under a single entry node (below threshold) and are not shown individually. That way the printed tree is still consistent in that all child nodes' values sum up to their parent's total, but unimportant nodes can be hidden.

Cost. Every first call to timer%start() at a certain point in the tree allocates a small amount of memory. If the timer is no longer needed, all that memory can be freed again with the %free() method.

Additionally, timer instances start and can be %disabled(), in which case almost all operations return immediately without (as much of) the overhead that would be necessary when doing the actual time measurements. That said, the actual overhead when the timer is enabled should be in the order of thousands of cycles per start%()/%stop() pair.

Due to this small overhead it is of course never advised to instrument completely down to the innermost loops. On a per-function level, though, the overhead should be small enough in most cases – otherwise your functions are problematically small anyway. It is of course possible to create multiple timer_t objects, that instrument the code in different granularity, and which do not need to be enabled all the time.

Availability. ftimings is open-source and is already available as a module on many of the clusters as well as on the Draco and Cobra installations at the MPCDF. It provides a pkg-config file with the usual -libs and -cflags arguments, as well as the Fortran specific flags in -variables=fcflags. Example usage:

```
#> module load gcc/8 ftimings
#> LDFLAGS="$(pkg-config ftimings-1-gcc-8 --libs) -Wl,-rpath=$FTIMINGS_HOME/lib"
#> FCFLAGS=$(pkg-config ftimings-1-gcc-8 --variable=fcflags)
#> gfortran $FCFLAGS foo.F90 -o foo $LDFLAGS
```

Note the peculiar name, ftimings-1-gcc-8, of the pkg-config file. This signifies the API version (1) of ftimings as well as the Fortran compiler used to build it – since Fortran modules are compiler dependent this is sadly necessary.

For codes that are already equipped with perflib instrumentation there is also a wrapper library that provides a compatible API of most of perflib's inter-

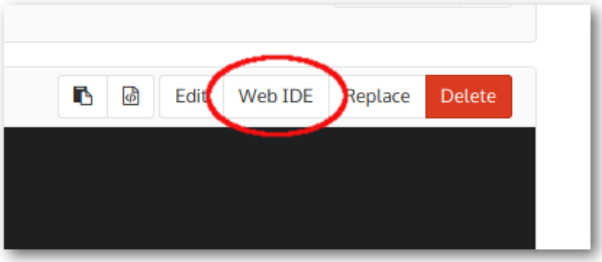
face. There are additional `pkg-config` files (e. g. `ftimings_perflib-1-gcc-8`) that provide the necessary linker arguments to link with that shim layer (Note that the focus of `perflib` is slightly different from `ftimings`', `perflib` has less overhead and strives to exclude that overhead from the resulting numbers. `ftimings`' focus is to provide hierarchical/nested time information).

The source code and documentation, also for the C API, can be found in the libraries repository, publicly available at our gitlab instance.

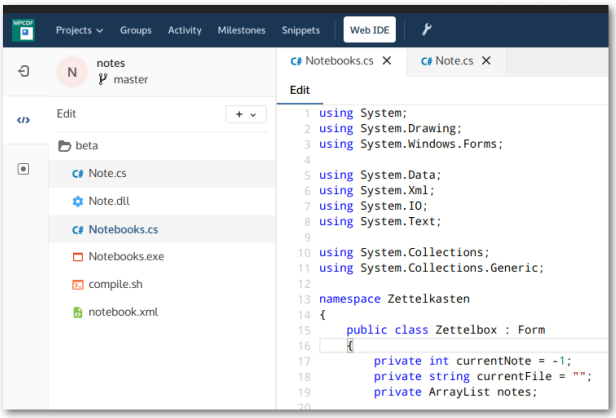
GitLab: Online Editing of Source Code

Thomas Zastrow

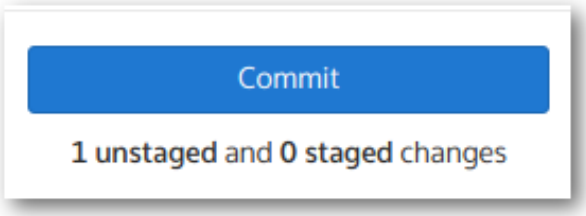
In addition to Git's basic versioning functionality, GitLab (`https://gitlab.mpcdf.mpg.de`) offers several other features which support collaboration and project management. The integrated WebIDE makes it now easy and comfortable to edit source code directly in the GitLab web application – without the need of checking out a Git project to your local computer. Once you have navigated to a source document, GitLab offers you to open it in the WebIDE (top right of the screen):



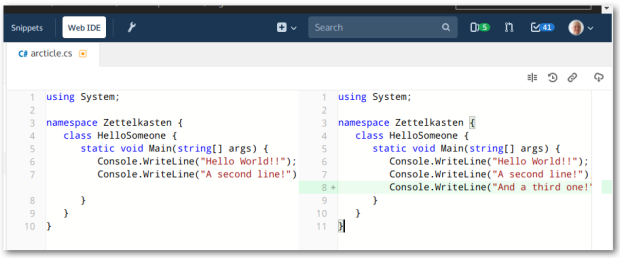
The WebIDE offers syntax highlighting and bracket matching for many modern programming languages like Java, C, PHP or Python. With the tab on the left, the user can navigate through the files of the current Git repository and open more than one file at the same time for editing. It is also possible to create new files and add them to the repository:



The WebIDE keeps track of all changes to existing or new files and offers to add and commit them (bottom left of the screen):



The WebIDE has a tight integration into GitLab's versioning functionality. For example, it can compare the current version of a file with its previous commit:



GitLab's WebIDE is far away from the functionality of well-known desktop based IDEs like Netbeans or Eclipse. There is no code completion, no refactoring and no integration of build systems. However, it allows the developer to do a quick change of their source code, without losing the functionality of a Git-based versioning and GitLab's collaboration tools. In combination with the continuous integration function of GitLab, it is also possible to implement a whole workflow without leaving the GitLab web interface.