UMAP Documentation

Release 0.0.3

Marty McFadden

Apr 11, 2023

BASICS

Umap is a library that provides an mmap()-like interface to a simple, user- space page fault handler based on the userfaultfd Linux feature (starting with 4.3 linux kernel). The use case is to have an application specific buffer of pages cached from a large file, i.e. out-of-core execution using memory map.

- Take a look at our Getting Started guide for all you need to get up and running with umap.
- If you are looking for developer documentation on a particular function, check out the code documentation.
- Want to contribute? Take a look at our developer and contribution guides.

Any questions? File an issue on GitHub.

CHAPTER

ONE

GETTING STARTED

This page provides information on how to quickly get up and running with umap.

1.1 Dependencies

At a minimum, cmake 3.5.1 or greater is required for building umap.

1.1.1 UMAP Build and Installation

The following lines should get you up and running:

```
$ git clone https://github.com/LLNL/umap.git
$ mkdir build && cd build
$ cmake -DCMAKE_INSTALL_PREFIX="<Place to install umap>" ../umap
$ make
$ make install
```

By default, umap will build a Release type build and will use the system defined directories for installation. To specify different build types or specify alternate installation paths, see the *Advanced Configuration*.

Umap install files to the `lib`, `include` and `bin` directories of the `CMAKE_INSTALL_PREFIX`.

1.1.2 Basic Usage

The interface to umap mirrors that of mmap(2) as shown:

The following code is a simple example of how one may use umap:

CHAPTER

TWO

ADVANCED CONFIGURATION

Listed below are the umap-specific options which may be used when configuring your build directory with cmake. Some CMake-specific options have also been added to show how to make additional changes to the build configuration.

```
cmake -DENABLE_LOGGING=Off
```

Here is a summary of the configuration options, their default value, and meaning:

These arguments are explained in more detail below:

- `ENABLE_LOGGING` This option enables usage of Logging services for umap. When this support is enabled, you may cause umap library to emit log files by setting the `UMAP_LOG_LEVEL` environment variable to "INFO" (for information-only logs), "WARNING" (for warning info logs), "ERROR" for (for errors only logs), and "DEBUG" for all debug messages to be emitted to a log file.
- `ENABLE_DISPLAY_STATS` When this option is turned on, the umap library will display its runtime statistics before unmap() completes.
- `ENABLE_TESTS` This option enables the compilation of the programs under the tests directory of the umap source code.
- `ENABLE_TESTS_LINK_STATIC_UMAP` This option enables the compilation of the programs under the tests directory of the umap source code against static umap library.

CHAPTER

THREE

RUNTIME ENVIRONMENT VARIABLES

The interface to the umap runtime library configuration is controlled by the following environment variables.

```
UMAP_PAGESIZE=$((2*4096)) your_program_that_uses_umap
```

The following environment varialbles may be set:

- `UMAP_PAGE_FILLERS` This is the number of worker threads that will perform read operations from the backing store (including read-ahead) for a specific umap region.

  Default: *std::thread::hardware_concurrency()*

- `UMAP_PAGE_EVICTORS` This is the number of worker threads that will perform evictions of pages. Eviction includes writing to the backing store if the page is dirty and telling the operating system that the page is no longer needed.

  Default: *std::thread::hardware_concurrency()*

- `UMAP_EVICT_HIGH_WATER_THRESHOLD` This is an integer percentage of present pages in the Umap Buffer that informs the Eviction workers that it is time to start evicting pages.

  Default: 90

- `UMAP_EVICT_LOW_WATER_THRESHOLD` This is an integer percentage of present pages in the Umap Buffer that informs the Eviction workers when to stop evicting.

  Default: 70

- `UMAP_PAGESIZE` This is the size of the umap pages. This must be a multiple of the system page size.

  Default: System Page Size

- `UMAP_BUFSIZE` This is the total number of umap pages that may be present within the Umap Buffer.

  Default: (90% of free memory)

- `UMAP_MONITOR_FREQ` This is the interval (in seconds) for the monitoring thread to print statistics, e.g., filled pages, free pages and processed events for debugging or tuning.

  Default: 0

CHAPTER

FOUR

SPARSE MULTI-FILES BACKING STORE INTERFACE

UMap provides a sparse and multi-files store object called "SparseStore", which partitions the backing file into multiple files that are created dynamically and only when needed.

A SparseStore object is instantiated in either "create" or "open" mode.

In "create" mode, the total region size, page size, backing directory path, and partitioning granularity need to be specified.

In "open" mode, the backing directory path needs to be specified, along with a "read_only" boolean option.

To instantiate and use a SparseStore object in "create" mode:

```
Umap::SparseStore * sparse_store;
sparse_store = new Umap::SparseStore(numbytes,page_size,root_path,file_size);

// set umap flags
int flags = UMAP_PRIVATE;

if (start_addr != nullptr)
 flags |= MAP_FIXED;

const int prot = PROT_READ|PROT_WRITE;

/* Map region using UMap, Here, the file descriptor passed to umap is -1, as we do not␣
↪start with mapping a file
  instead, file(s) will be created incrementally as needed using the "SparseStore"␣
↪object. */

region = umap_ex(start_addr, numbytes, prot, flags, -1, 0, sparse_store);
```

To instantiate and use a SparseStore object in "open" mode:

```
Umap::SparseStore sparse_store;
bool read_only = true;
sparse_store = new Umap::SparseStore(root_path,read_only);

// set umap flags
int flags = UMAP_PRIVATE;

if (start_addr != nullptr)
 flags |= MAP_FIXED;

const int prot = PROT_READ|PROT_WRITE;
```

(continues on next page)

(continued from previous page)

```
region = umap_ex(start_addr, numbytes, prot, flags, -1, 0, sparse_store);
```

To unmap a region created with SparseStore, the SparseStore object needs to explicitely close the open files and then be deleted:

```
 // Unmap region
 if (uunmap(region, numbytes) < 0) {
     // report failure and exit
 }

 int sparse_store_close_files = store->close_files();
 if (sparse_store_close_files != 0 ){
     // report failure and exit
 }
 delete store;
}
```

CHAPTER

FIVE

INTEGRATION WITHIN CALIPER

UMap can be integrated into the Caliper: A Performance Analysis Toolbox in a Library for providing page access pattern profiling. We describe the steps for leveraging UMap in Caliper as follows:

- Install Caliper from https://github.com/LLNL/Caliper.git to <CALIPER_INSTALL_PATH>
- Build UMap by running cmake with -Dcaliper_DIR=<CALIPER_INSTALL_PATH>/share/cmake/caliper
- An example application is provided in /tests/caliper_trace. Make sure it is compiled
- Enable the page fault tracing before running the program to be profiled as follows

```
export CALI_SERVICES_ENABLE=alloc,event,trace,recorder
export CALI_ALLOC_TRACK_ALLOCATIONS=true
export CALI_ALLOC_RESOLVE_ADDRESSES=true
```

- This should produce a .cali output file with an automatically generated filename, e.g., "200611-155825_69978_QSZC2zryxwRh.cali".

Now we describe how to analyze the profiling results. To simply print all records captured from Caliper, simply use:

```
cali-query -t <filename>
```

For advanced queries, e.g. count the number of page faults per memory region:

```
cali-query -q "select alloc.label#pagefault.address,count() group by alloc.label
↪#pagefault.address where pagefault.address format table" <filename>
```

CHAPTER

SIX

CONTRIBUTION GUIDE

This document is intented for developers who want to add new features or bugfixes to umap. It assumes you have some familiarity with git and GitHub. It will discuss what a good pull request (PR) looks like, and the tests that your PR must pass before it can be merged into umap.

6.1 Forking umap

If you aren't an umap deveolper at LLNL, then you won't have permission to push new branches to the repository. First, you should create a fork. This will create a copy of the umap repository that you own, and will ensure you can push your changes up to GitHub and create pull requests.

6.1.1 Developing a New Feature

New features should be based on the `develop` branch. When you want to create a new feature, first ensure you have an up-to-date copy of the `develop` branch:

```
$ git checkout develop
$ git pull origin develop
```

You can now create a new branch to develop your feature on:

```
$ git checkout -b feature/<name-of-feature>
```

Proceed to develop your feature on this branch, and add tests that will exercise your new code. If you are creating new methods or classes, please add Doxygen documentation.

Once your feature is complete and your tests are passing, you can push your branch to GitHub and create a PR.

6.1.2 Developing a Bug Fix

First, check if the change you want to make has been fixed in `develop`. If so, we suggest you either start using the `develop` branch, or temporarily apply the fix to whichever version of umap you are using.

If the bug is still unfixed, first make sure you have an up-to-date copy of the develop branch:

```
$ git checkout develop
$ git pull origin develop
```

Then create a new branch for your bugfix:

```
$ git checkout -b bugfix/<name-of-bug>
```

First, add a test that reproduces the bug you have found. Then develop your bugfix as normal, and ensure to `make test` to check your changes actually fix the bug.

Once you are finished, you can push your branch to GitHub, then create a PR.

6.1.3 Creating a Pull Request

You can create a new PR here. Ensure that your PR base is the `develop` branch of umap.

Add a descriptive title explaining the bug you fixed or the feature you have added, and put a longer description of the changes you have made in the comment box.

Once your PR has been created, it will be run through our automated tests and also be reviewed by umap team members. Providing the branch passes both the tests and reviews, it will be merged into umap.

6.1.4 Tests

Umap uses Bamboo for continuous integration tests. Our tests are automatically run against every new pull request, and passing all tests is a requirement for merging your PR. If you are developing a bugfix or a new feature, please add a test that checks the correctness of your new code.

Umap's tests are all in the `tests` directory.