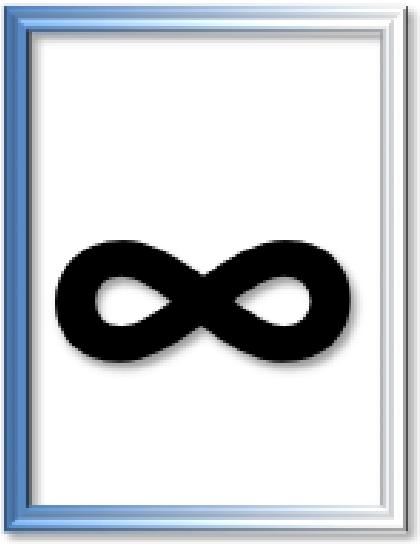
amask v1.0

AMASK: Affinity MASKs for parallel processes/threads
Document Revision 1.0
May 3, 2017



Kent Milfeld
milfeld@tacc.utexas.edu

High Performance Computing
Texas Advanced Computing Center
The University of Texas at Austin

Abstract

Amask is a set of tools for application developers and users to discover the affinity masks of application processes (MPI ranks or OpenMP threads) so they can determine where the processes can run. Amask has the following components:

- Stand-alone executables to report default masks of OpenMP, MPI or Hybrid executions in an interactive or batch environment.
- API for instrumenting applications to report affinity masks from within a program.
- Utilities: timers, set process/thread affinity, create loads (for `top` viewing)

Our intention is to create a tool that provides simple-to-understand affinity information. Bug reports and feedback on usability and improvements are welcome; send to milfeld@tacc.utexas.edu with amask in the subject line.

If you use amask, cite:

github.com/TACC/amask, "amask: Affinity Mask", Texas Advanced Computing Center (TACC), Kent F. Milfeld. [1]

Contents

1 Installation **2**

2 Process Thread Affinity Mask for a Parallel Execution **3**

3 Using the amask Library **7**

3.1 Get Masks Inside a Program 7

3.2 Useful Utilities 8

3.3 Other things you should know 9

References 9

1 Installation

Amask is easy to build. Execute `make` to build the stand-alone executables and a library for those who want to instrument their application.

Download the github repository[1] by clicking on the "Download ZIP" file, and expanding it in a convenient location.

```
unzip  amask-master.zip
```

You can also clone the git repository:

```
git clone https://github.com/TACC/amask
```

This will create a top level directory called `amask`, with subdirectories `docs` and `src`. Change directory to `amask`. Edit the `Makefile` to include an appropriate MPI compiler and OpenMP flag (the defaults are for Intel). Execute `make`.

The executables will be placed in the `amask/bin` directory and the library will be placed in `amask/lib`. Include `.../amask/bin` in your `PATH` variable.

[1] https://github.com/TACC/amask

2 Process Thread Affinity Mask for a Parallel Execution

Execute one of the stand-alone executables, `omp_affinity`, `mpi_affinity`, or `hybrid_affinity` in an OpenMP, MPI or hybrid environment, respectively, to obtain the expected affinity mask for each process/thread for your program in the same environment. That is, before you execute your program, execute the appropriate stand-alone affinity program to get a listing of the masks for each OpenMP thread and/or MPI process that will occur for your program execution (provided the application doesn't adjust affinity).

In Listing 2.1 a batch job runs the `mpi_affinity` executable before running an application, to observe the affinity mask for each rank that the application (`my_mpiapp`) will have. By removing the `my_mpiapp` execution line and requesting less time for the batch job, you can quickly discover the default affinity mask for the parallel environment. You can adjust affinity environment variables and quickly assess their impact on the affinity masks for your application.

Listing 2.1: Listing affinity masks for MPI environment

```
1
2       #!/bin/bash
3       #SLURM  -n 16 -N 1
4           ...
5       #Batch Script for TACC machine
6          ...
7        mpirun ./mpi_affinity    #ibrun ./mpi_affinity #@TACC
8        mpirun ./my_mpiapp       #ibrun ./my_mpiapp    #@TACC
```

If a site allows users interactive access to batch nodes (see idev utility @TACC), then the amask commands can be run interactively. Listing 2.2 shows interactive executions to discover the affinity masks that any pure OpenMP, pure MPI or a hybrid application would have for the environment. (The environment includes the OMP variables, number of MPI tasks requested, and MPI affinity environment variables set in the mpi launcher: mpirun, or ibrun at TACC.)

Listing 2.2: Listing OpenMP/MPI and Hybrid masks

```
1
2       export OMP_NUM_THREADS=2; omp_affinity   # pure OpenMP
3
4       mpirun -np 4 mpi_affinity                # pure MPI
5
6       export OMP_NUM_THREADS=2; mpirun -np 4 hybrid_affinity   # Hybrid
```

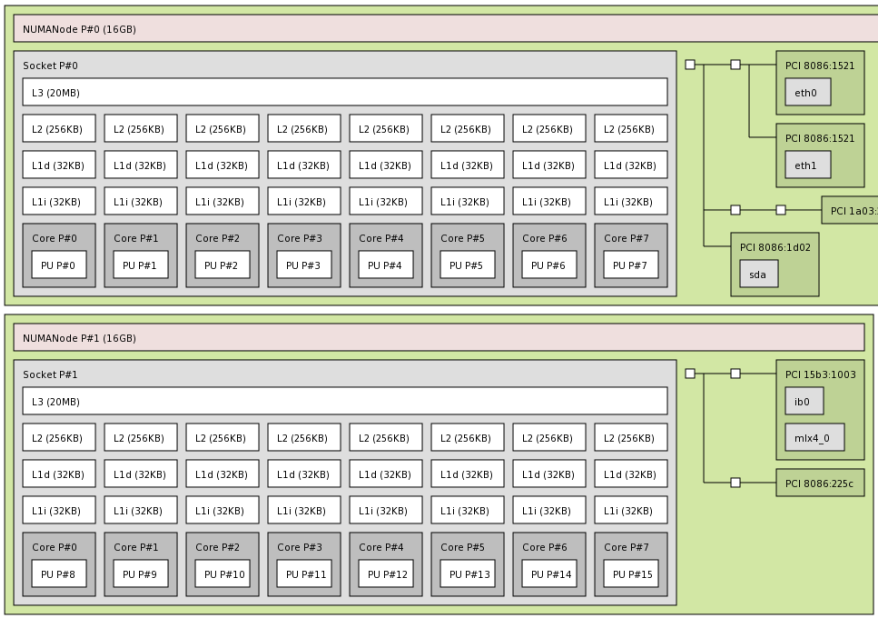


Figure 2.1: Hardware information from lstopo

How to read the output:

The output in Listing 2.3 is for an 8-rank MPI execution on a platform with 2 sockets, 8 cores/socket, no hyperthreading, and processor ids 0-7 on socket 1 and 8-15 on socket 2. One can derive these details from `lstopo`. Figure **??** shows an `lstopo` report for this system (Sandy Bridge compute node on the TACC Stampede machine.) The same information can be extracted from the contents of the `/proc/cpuinfo` file. The `lscpu` utilitly provides the number of sockets, cores and SMT threads, without proc-id assignments.

The MPI processes are executing on processors (cores) 0-7. The rows of the matrix are labeled by the MPI ranks and the columns represent the machine's processor-ids (`Core P#` in 2.1) from 0 to Nprocs-1, where Nprocs is the number `PUs` from `lstopo` or the number of processors reported in `/proc/cpuinfo`. For this run there are 8 rows, one for each of the 8 MPI ranks (0000 through 0007). Since there are 16 processors (Nprocs), there are 16 character locations in each row, representing processors 0 - 15. Numbers in a row position mean that the mask bit is set for the the corresponding processor (for the MPI rank, process, listed at the beginning of the row can execute there). When multiple numbers are in a row (multiple bits set) it means that the rank can run (float) on any of those processors. (The - characters indicate the mask bits are not set.) The processor id number is determined by adding the digit in the row to the number directly above in the header. (This quirky approach allows us to represent a processor's mask as a set of single characters (digits) for each bit, and allows one to determine the processor id (proc-id) number by a "look up"– adding the digit in the row to header value above).

Listing 2.3: Default MPI Environment

```
$ # 8 tasks requested, MVAPICH2 MPI

$ mpirun mpi_affinity   # non-TACC
$ ibrun  mpi_affinity   # @TACC


      Each row of matrix is an Affinity mask.
      A set mask bit = matrix digit + column group # in |...|
rank |    0    |   10    |
0000 0---------------
0001 -1--------------
0002 --2-------------
0003 ---3------------
0004 ----4-----------   #digit + group # in header = proc-id
0005 -----5----------
0006 ------6---------
0006 -------7--------
```

The output in Listing 2.4 is for the same platform as 2.3, but the affinity environment has been changed with the `tacc_affinity` script on the ibrun launcher line (`ibrun tacc_affinity mpi_affinity`). There are other ways to change the affinity, such as using another launcher, compiling with a different mpi library, or setting MPI environment variables. In this case `tacc_affinity` positions the first 4 tasks on socket 0, and allows each rank to float on any of the processors (cores) of the socket, by setting the first 8 bits of the mask for each of the 4 ranks, 0-3. Ranks 4-7 execute on the second socket and are allowed to float across all cores (proc-ids 8-15) of the socket. (`tacc_affinity` is not part of the amask utilities.)

Listing 2.4: TACC Tailored Environment through tacc-affinity

```
$ # 8 tasks requested, MVAPICH2 MPI
$ # + tacc_affinity -- sets MPI affinity @TACC

$ ibrun tacc_affinity mpi_affinity   # @TACC

      Each row of matrix is an Affinity mask.
      A set mask bit = matrix digit + column group # in |...|
rank |   0     |   10    |
0000 01234567--------       # -- mask to run anyplace on socket1
0001 01234567--------       # rank 1 can execute on any of 0-7 cores
0002 01234567--------       # rank 2 can execute on any of 0-7 cores
0003 01234567--------       # ...
0004 --------89012345       # -- mask to run anyplace on socket2
0005 --------89012345       # rank 5 can execute on any of cores 8-15
0006 --------89012345       # Bits are for proc-ids (cores)  8, 9,
    10-15
0007 --------89012345       # Read as 8+0, 9+0, 0+10, 0+10, ... 10+15
```

A more challenging situation is the determination of the masks on a system with hundreds of hardware threads, such as the Intel KNL processor. At TACC, Stampede2 has 4,200 PHI 7250 processors (KNLs), each with 272 hardware threads (68 cores x 4 SMT threads/core). Before version 1.0 amask would

display rows of 272 proc-ids (processor ids=hardware threads); but that became a bit unwieldy for users to display on laptops. Now, by default, a matrix of core-ids versus process-ids is displayed. If there are n SMTs/core, there will be n rows for each process (rank/thread-id) listed (on the left). For the KNL described above there will be 4 rows displayed for each process (corresponding to proc-id 0-67, 68-135, 136-203, and 204-272) with the header displaying core-ids. This is shown in Listing 2.5.

Listing 2.5: Process masks for an 8-thread execution on 68-core KNL

```
1
2  $ export OMP_NUM_THREADS=8
3  $ export OMP_PROC_BIND=close8
4  $ ./omp_affinity
5
6        Each row of matrix is an Affinity mask. A set mask bit = matrix digit +
             column # in |...|
7  thrd |    0    |   10    |   20    |   30    |   40    |   50    |   60    |
8  0000 0=========================================================================
9       -------------------------------------------------------------------------
10      -------------------------------------------------------------------------
11      -------------------------------------------------------------------------
12 0001 =========================================================================
13      0------------------------------------------------------------------------
14      -------------------------------------------------------------------------
15      -------------------------------------------------------------------------
16 0002 =========================================================================
17      -------------------------------------------------------------------------
18      0------------------------------------------------------------------------
19      -------------------------------------------------------------------------
20 0003 =========================================================================
21      -------------------------------------------------------------------------
22      -------------------------------------------------------------------------
23      0------------------------------------------------------------------------
24 0004 =1=======================================================================
25      -------------------------------------------------------------------------
26      -------------------------------------------------------------------------
27      -------------------------------------------------------------------------
28 0005 =========================================================================
29      -1-------------------------------------------------------------------------
30      -------------------------------------------------------------------------
31      -------------------------------------------------------------------------
32 0006 =========================================================================
33      -------------------------------------------------------------------------
34      -1-------------------------------------------------------------------------
35      -------------------------------------------------------------------------
36 0007 =========================================================================
37      -------------------------------------------------------------------------
38      -------------------------------------------------------------------------
39      -1-------------------------------------------------------------------------
```

This display is basically core occupation – something users found they would rather see than 272-character masks. The proc-id numbers for the 2nd, 3rd and 4th row can be obtained in the usual manner, but 68, 136, and 204 must be added to the respective row. When you need to work directly with the mask and easily determine the proc-ids, it is much easier to use the mask display by using the `-ls` option on command line.

3 Using the amask Library

The basic operations used for reporting masks by the `amask` executables were collected into a library, so that users could instrument there own applications to display the masks of MPI processes and OpenMP threads.

3.1 Get Masks Inside a Program

To report the masking from inside a program, include the amask API routine, `omp_report_mask()`, `mpi_report_mask()`, or `hybrid_report_mask()`, within an OpenMP parallel region, after MPI has been initialized, or within an OpenMP region of a hybrid program, respectively. These functions don't require any arguments or include files:

- omp_report_mask()
- mpi_report_mask()
- hybrid_report_mask()

However, to report masks for hybrid code, it may be necessary to initialized MPI with the `MPI_Init_thread()` routine. View the `amask` codes to see how easy it is to include them in your own program. The snippets below show how they are to be used:

Listing 3.1: Invoking mask report inside code

```
1  // Pure OpenMP code
2     #pragma omp parallel
3     {
4        omp_report_mask();
5      ...
6     }
7
8  // Pure MPI code
9     MPI_Init(NULL,NULL);
10
11       mpi_report_mask();
12    ...
13    MPI_Finalize();
14
15 // Hybrid code
16    MPI_Init_thread(NULL,NULL, MPI_THREAD_MULTIPLE, &provided);
17
18    mpi_report_mask();  // helpful: reports ONLY MPI process masks
19
20   #pragma omp parallel
21   {
22      hybrid_report_mask();
23      ...
24   }
25   ...
26   MPI_Finalize();
```

3.2 Useful Utilities

- load_cpu_nsec(nsec) — Puts load on process/thread for nsec seconds
- map_to_procid(proc-id) — Sets process/thread to execute on proc-id
- gtod_timer() — Easy to use Get Time of Day clock
- tsc() — Returns time stamp counter value

A thread or process that calls `load_cpu_nsec(int nsec)` will execute integer operations (a load) for nsec seconds. nsec must be zero or a positive integer. Use the `cmdln_get_nsec_or_help(int * nsec, int argc, char * argv[])` function to extract an integer from the command line for nsec (e.g. `mpi_affinity 10`).

A thread or process that calls `map_to_procid(int proc-id)` will assign the calling thread or process to execute on proc-id, by setting the appropriate bit in the scheduling mask. For example, in a parallel region the unique thread-id returned from `omp_get_thread_num()` can be used in an arithmetic operation (linear, modulo, etc.) to create a unique proc-id to be executed on.

The function `gtod_timer()` returns a double precision number with the number of wall-clock seconds since the previous call. The first call sets the time to zero. The function uses the Unix `gettimeofday` utility, and can be called from C/C++ and Fortran. See comments in the code for more details.

The function `tsc()` returns an 8-byte integer (Unix) time stamp count from the `rdtsc` instruction. Use this to capture the difference between the counts for determining the cost a of a small set of operations (instructions, code statements). This is not meant to be used as as a general timer, since the processor frequency may change.

3.3 Other things you should know

Build `amask` with the same MPI library that your application will use. On systems with multi-vendor MPI libraries, build amap for each vendor version you will use. With IMPI (Intel MPI) you can set I_MPI_DEBUG=4 to separately report the mask (and other information) for each MPI process.

Bibliography

[1] Amask is a set of executables and routines for reporting affinity information. https://github.com/tacc/amask