# SYMBOL VERSIONING FOR OPEN-MPI

Author: Alastair McKinstry <mckinstry@debian.org>

Version: 0.1

Date: 2016-08-10

### Summary

This paper proposes to use symbol versioning in Open-MPI in order to facilitate upgrades and development. The status quo expects a “rebuild everything on new version” policy, which while accepted practice in HPC, is non-scalable and impedes using MPI in wide use in multi-core systems with deep library heirarchies. Symbol versioning has been the standard practice in base libraries such as libc, in ELF binaries on most Unix-based OS for 20 years.

### Background

What is Symbol versioning

Symbol versioning is a described here: <https://gcc.gnu.org/wiki/SymbolVersioning>

In short, each object compiled into a dynamic library has an optional symbol attached. These are added as “@@VERSION” suffixes to a symbol name. This can be seen using “readelf”:

debian:~$ readelf --syms /lib/x86\_64-linux-gnu/libc.so.6 | more

Symbol table '.dynsym' contains 2245 entries:

Num: Value Size Type Bind Vis Ndx Name

0: 0000000000000000 0 NOTYPE LOCAL DEFAULT UND

1: 000000000001f890 0 SECTION LOCAL DEFAULT 13

2: 00000000003977c8 0 SECTION LOCAL DEFAULT 22

3: 0000000000000000 0 OBJECT GLOBAL DEFAULT UND \_rtld\_global@GLIBC\_PRIVATE (26)

4: 0000000000000000 0 OBJECT GLOBAL DEFAULT UND \_\_libc\_enable\_secure@GLIBC\_PRIVATE (26)

5: 0000000000000000 0 FUNC GLOBAL DEFAULT UND \_\_tls\_get\_addr@GLIBC\_2.3 (27)

6: 0000000000000000 0 OBJECT GLOBAL DEFAULT UND \_rtld\_global\_ro@GLIBC\_PRIVATE (26)

7: 0000000000000000 0 FUNC GLOBAL DEFAULT UND \_dl\_find\_dso\_for\_object@GLIBC\_PRIVATE (26)

8: 0000000000000000 0 NOTYPE WEAK DEFAULT UND \_dl\_starting\_up

9: 0000000000000000 0 OBJECT GLOBAL DEFAULT UND \_dl\_argv@GLIBC\_PRIVATE (26)

10: 0000000000069130 349 FUNC GLOBAL DEFAULT 13 putwchar@@GLIBC\_2.2.5

11: 000000000008ce90 32 FUNC GLOBAL DEFAULT 13 \_\_strspn\_c1@@GLIBC\_2.2.5

12: 00000000000f6db0 16 FUNC GLOBAL DEFAULT 13 \_\_gethostname\_chk@@GLIBC\_2.4

13: 000000000008ceb0 37 FUNC GLOBAL DEFAULT 13 \_\_strspn\_c2@@GLIBC\_2.2.5

14: 000000000010d1d0 192 FUNC GLOBAL DEFAULT 13 setrpcent@@GLIBC\_2.2.5

15: 000000000009ece0 10 FUNC GLOBAL DEFAULT 13 \_\_wcstod\_l@@GLIBC\_2.2.5

16: 000000000008cee0 70 FUNC GLOBAL DEFAULT 13 \_\_strspn\_c3@@GLIBC\_2.2.5

17: 00000000000e7840 33 FUNC GLOBAL DEFAULT 13 epoll\_create@@GLIBC\_2.3.2

18: 00000000000d0530 33 FUNC WEAK DEFAULT 13 sched\_get\_priority\_min@@GLIBC\_2.2.5

When compiling against a library, if no symbols are specified in the library, versioning is ignored. If the library provides symbols, the symbol version is recorded.

Notice that the version is that of the *symbol* not the library, so libc 2.23 ships multiple versions of some symbols:

73: 000000000016b7fc 4 OBJECT GLOBAL DEFAULT 16 \_sys\_nerr@@GLIBC\_2.12

74: 000000000016b808 4 OBJECT GLOBAL DEFAULT 16 \_sys\_nerr@GLIBC\_2.4

75: 000000000016b804 4 OBJECT GLOBAL DEFAULT 16 \_sys\_nerr@GLIBC\_2.3

76: 000000000016b800 4 OBJECT GLOBAL DEFAULT 16 \_sys\_nerr@GLIBC\_2.2.5

The “@@” shows that 2.12 is the current default, while previous versions are only used for those binaries that were linked against them.

e.g.

debian$ gcc hello.c -o hello

#include <stdio.h>

#include <mpi.h>

int main (argc, argv)

int argc;

char \*argv[];

{

int rank, size;

MPI\_Init (&argc, &argv); /\* starts MPI \*/

MPI\_Comm\_rank (MPI\_COMM\_WORLD, &rank); /\* get current process id \*/

MPI\_Comm\_size (MPI\_COMM\_WORLD, &size); /\* get number of processes \*/

printf( "Hello world from process %d of %d\n", rank, size );

MPI\_Finalize();

return 0;

}

debian$ readelf –syms ./hello

…

4**8: 0000000000000000 0 FUNC GLOBAL DEFAULT UND printf@@GLIBC\_2.2.5**

49: 00000000004008d0 2 FUNC GLOBAL DEFAULT 14 \_\_libc\_csu\_fini

50: 0000000000400700 42 FUNC GLOBAL DEFAULT 14 \_start

51: 0000000000000000 0 NOTYPE WEAK DEFAULT UND \_\_gmon\_start\_\_

52: 0000000000000000 0 NOTYPE WEAK DEFAULT UND \_Jv\_RegisterClasses

53: 00000000004008d4 0 FUNC GLOBAL DEFAULT 15 \_fini

54: 0000000000000000 0 FUNC GLOBAL DEFAULT UND \_\_libc\_start\_main@@GLIBC\_

**55: 0000000000000000 0 FUNC GLOBAL DEFAULT UND MPI\_Init**

56: 0000000000000000 0 NOTYPE WEAK DEFAULT UND \_ITM\_deregisterTMCloneTab

57: 00000000004008e0 4 OBJECT GLOBAL DEFAULT 16 \_IO\_stdin\_used

…

Here, glibc provides a versioned printf, so its version “GLIBC\_2.2.5” is recorded, but MPI\_Init provides no version. When linking the executable ‘./hello’ at run-time, the linker looks for a specific version of printf (GLIBC\_2.2.5) to use, but will use whatever version of MPI\_Init it finds in the specified libraries.

If a new, incompatible version of “printf” is supplied in a revision of libc, it will have a new symbol, and the default symbol would change, but this binary would continue to use version “GLIBC\_2.2.5” and hence not break. This allows the API and ABI to change but existing binaries continue to work.

Note that if the API changes, compilation is no longer guaranteed: e.g. if printf were to change signature, the program would break until the code is ported, but existing binaries continue to work.

Implementation

Symbol versioning is implemented using a map file, specifying the version number for symbols:

e.g.

debian$ cat foocode.c

int int\_add (int a, int b) {

return a+b;

}

int foo( int a) {

return a+2;

}

int bar (int b) {

return a+3;

}

int bax (int a, int b) {

return int\_add(a, b) + 4;

}

$ cat foo.ver

FOO\_1.0 {

global:

foo;

bar;

local:

\*;

};

FOO\_1.1 {

global:

bax;

} FOO\_1.0;

The library is linked then:

$ gcc -o libfoo.so.1 -shared -Wl,--version-script=foo.ver

This results in foo and bar having version FOO\_1.0, bax having version FOO\_1.1 and any other objects not being visible, due to the ‘local’ clause.

Now, it is trivially possible to add new functionality to libfoo.so without breaking existing binaries,

but changing binaries can be done with versioned symbols.

For example, if we wish to make bax() operate on doubles:

double bax (double a, double b) {

return a+b+4:

}

Both the API and ABI break. However, symbol versioning enables us to ship two versions of bax(), solving this problem:

int bax\_v1 (int a, int b) {

return int\_add (a, b) + 4;

}

double bax\_v2 (int a, int b) {

return a + b + 4;

}

\_\_asm\_\_(".symver original\_foo,foo@");

\_\_asm\_\_(".symver bax\_v1, bax@VERS\_1.1");

\_\_asm\_\_(".symver bax\_v2 ,bax@@VERS\_1.2");

Publically, libfoo.h might contain:

#ifdef LIBFOO\_H

double bax( double a, double b);

int foo (int a);

int bar (int a);

#endif

Notice that while the API breaks, and any existing code using the library will cease to compile and need to be ported, all existing binaries will continue to work. This is essential for upgrade-in-place, but also testing and development of deep

### Proposal

\* Add versioning maps to the public libraries libopenmpi.so.20 and liboshmem.so.20.

\* The major version numbers of the public libraries be held at 20 for the 2.x series, i.e. no functionality be dropped.

\* The minor version number be incremented on any API/ABI changes. Where symbol signatures change, the methods described above and in the GNU ‘ld’ documentation be used. ie. the next ABI change for OpenMPI should then ship (for example) libopenmpi.so.20.1.0

This change would immediately have no operational impact on OpenMPI, except for a negligible performance gain in linking, due to the linking lookup structures being smaller as “local” symbols are removed (a significant startup gain is seen on C++ programs such as LibreOffice, QT due to this).

However, it does impose an engineering overhead in that the symbols version table needs to be updated on new releases, along with the above “glue” for older symbols.

### Platform support

Symbol versioning is a feature of the ELF binary format and is available on Linux, BSD, Solaris and HPUX Operating Systems, but not Mac OSX (MACH format) or AIX (DWARF).

Equivalent functionality may be achieved on OSX using compatibility if the “minor” versioning is used, incremented each time, while holding the major library version constant.

For Mac OS, it is recommended that the library then contains both the major and minor numbers in the name, ie.

\* libmpi.20.0.dylib

\* libmpi.dylib → libmpi.20.0.dylib

### MOTIVATION AND ALTERNATIVES

The standard alternative approach is to have multiple OpenMPI implementions in different directory trees:

/opt/openmpi-1.10.3

/bin/…

/lib/...

/opt/openmpi-2.20

/bin/..

/lib/…

This is often done on HPC systems, typically with the use of ‘modules’ to select the appropriate implementation. Applications are then linked against these.

This breaks down when there are multiple libraries, and levels of libraries, built on top. Currently the dependency tree in Debian Linux is 5-deep, eg. ParaView → libadios → libnetcdf → libhdf5 → libopenmpi. This will only get worse at MPI is embedded deeper in multicore systems; hundreds to thousands of applications and libraries will need to be rebuilt for any OpenMPI release change, and bugfixes become intolerably difficult.

### Packaging in Debian-like distributions

Debian (and other distributions) use packages to encapsulate file conflicts when installing software. No two packages may ship the same file (eg. /usr/lib/libopenmpi.so.20.0.0) Thus a library like OpenMPI would typically have two packages: *libopenmpi20* shipping /usr/lib/libopenmpi.so.20.0.0, and *libopenmpi-dev*, containing e.g /usr/include/mpi.h. In the event of a new ABI-changing version (SOVER 21), a new incompatible libopenmpi-dev package will be uploaded, but the shared library would be in *libopenmpi21,* enabling both old and new libraries to co-exist. The software can then be incrementally upgraded or rebuilt using the new library.

It is anticipated that the private libraries (e.g. libopen-rte.so.20.\*) will ship in a separate package with the binaries, *openmpi-bin*. There will only be one version of *openmpi-bin* installed at any time. On upgrade to *libopenmpi21*, the new version of *openmpi-bin* will use *libopenmpi21,*but all other binaries using *libopenmpi20*  will continue to work.