Application Development and Readiness for Sierra: An MPI Challenge

Sandia National Laboratories

James Elliott

jjellio@sandia.gov

CCR
Center for Computing Research

Sandia National Laboratories is a multimission laboratory managed and operated by [illegible]ational Technology & Engineerin[illegible]lutions of S[illegible]ndia, LLC, a wholly owned [illegible]ary of [illegible]oney[illegible]ell International Inc. [illegible]S. [illegible]epar[illegible]men[illegible] of Energy's Na[illegible]ear [illegible]ecur[illegible]ty Administration under [illegible]ct DE-NA[illegible]003[illegible]25. (SAND2020-[illegible]59 PE)

Unclassified Unlimited Release

Outline

- An evolution of MPI implementations
- An API left behind: how developers and *users* pay the price
  - Misunderstood performance gaps

    An API's ignorance leaves blood stains on the command line
  - Well documented semantics? (Cue laughter)

    Enjoy your deadlocks and correctness problems
  - Convoluted code paths depending on compile-time/runtime parameters

    All because an API refuses to provide introspection
- What can be done about it?

An evolution of MPI implementations

MPI_Send(**const void *buf**, int count, MPI_Datatype dtype, dest, tag, MPI_Comm comm)

- Was well defined – but has assumption that **buf** is addressable
- Device-aware MPI allows **buf** to be a device, host, or 'managed' pointer
- MPI implementations *have* evolved, but not the API
  - New runtime parameters
  - **Poorly documented** implementation details (that have leaked out)
  - Hard to vet if you are **using the MPI implementation correctly**

Which internal code path should you be following?
<u>It needs to match the runtime settings</u>!

Center for Computing Research

An API left behind, and how developers and *users* pay the price

MPI_Send(**const void *buf**, int count, MPI_Datatype dtype, dest, tag, MPI_Comm comm)

Device-aware MPI allows this **buf** to be a device, host, or 'managed' pointer

Different platforms may offer varied levels of support for each

- **App teams develop on diverse machines … targeting a future/different machine**
  - Early MPI implementations may not be complete or performant
  - Multi-platform compatibility requires supporting various combinations

    E.g., support CUDA before MPI implementations allow device pointers (Jaguar/Titan)
  - Customers using your application may need varied support

Convoluted code paths, exacerbated because MPI provides no way to query where **buf** can point.

Perhaps,

```
// Which memory spaces does this implementation support?
MPI_Get_memory_spaces( … )
// The expected cost to use memory space(s)
MPI_Get_memory_space_priorities( … )
```


Center for Computing Research

An API left behind, and how developers and *users* pay the price

MPI_Send(**const void *buf**, int count, MPI_Datatype dtype, dest, tag, MPI_Comm comm)

New runtime parameters … **blood stains on the command line**

‘cuda aware’ MPI=ON|OFF (better know **which state is the default**!)

- **Complexity of device-aware MPI leaks up to the user and runtime**
  - Various machines can have different default behaviors (device aware ‘on’ or ‘off’)
  - Software adds compile time checks to control default behavior
    - Assumes compile-time environment matches runtime! (**control your laughter**!)
  - Developers are savvy, make code paths runtime tunable (app will use device buffers or host)
    - App now depends on runtime settings, and setting needs to match MPI’s runtime settings
    - (**What could possibly go wrong?**)
  - Users may get away with ignorance, but see unexpected performance

Issue would be improved if MPI provided a way to express it’s current runtime settings.
(In a portable way!)

Would be nice if the app could decide if it wants to be ‘device aware’ or not API… ?

CCR
Center for Computing Research

An API left behind, and how developers and *users* pay the price

MPI_Send(**const void *buf**, int count, MPI_Datatype dtype, dest, tag, MPI_Comm comm)

Poorly documented implementation details

No memory allocations before MPI_Init!

- **Hidden semantics** introduced
  - Location of **buf** may require different code paths (for the MPI implementation)

    if (loc(buf) == Host) call ole_fashion(); else if (loc(buf) == …) call me_maybe();
  - Most device-aware implementations tend to do some form of tracking allocations and caching locations ('loc()' may be expensive, vendors may do more optimizations)

    Calling an MPI function with a buffer it hasn't tracked causes ~~imminent death~~ issues

    Effect: Tracking starts with MPI_Init()

    "The MPI standard does not say what a program can do before an MPI_INIT or after an MPI_FINALIZE."

    MPICH documentation for MPI_Init
  - When problems happened, was hard to diagnose (obscure segfaults *later* in program execution!)

    Did not always observe a crash in/near MPI_Init… instead, observed memory corruption

If MPI_Init is required to be first, then document it!
<u>Harder problems</u> are different code paths around MPI calls

Center for Computing Research

An API left behind, and how developers and *users* pay the price

MPI_Send(**const void *buf**, int count, MPI_Datatype dtype, dest, tag, MPI_Comm comm)

Hard to vet if you are using the MPI implementation correctly

Am I following all of the rules?

- **Compile time and runtime parameters – is the app following the rules?**
  - **Example:** Observed obscure correctness problems

    Appeared app was seeing 'old' data in MPI_Recv

    Valgrind clean / compiler warnings clean (**kitchen sink reported everything was fine!**)
  - Problem was one spot in the code that started passing managed memory (UVM) to MPI, but app did not enable 'cuda-aware'

    UVM is technically addressable on both device and host (we learned it is a 'cuda-aware feature')

    (Thanks Dave Richards and Ian Karlin!)
  - Implemented a PMPI profiler that tested/tracked all buffers into MPI and reported locations

    Quickly identified sources of problems … (fixed a single typedef)

    Fixed a few other locations in Trilinos…

How to solve this in a portable way?

Surely, new machines will have *some* new rules … (good discussion point!)

Center for Computing Research

What can be done about all of this?

MPI_Send(**const void *buf**, int count, MPI_Datatype dtype, dest, tag, MPI_Comm comm)

This talk has pointed out some naïve ways the MPI standard could potentially address these issues, **but would those techniques ever be portable**?

Should the standard adopt some some incarnation of a memory space?

MPI_Send(**const void *buf**, **MPI_Memory_Space location**,

int count, MPI_Datatype dtype, dest, tag, MPI_Comm comm)

Center for Computing Research

What can be done about all of this?

Instead… or, perhaps, in addition:

Many codes wrap MPI already – is it time for a communication portability layer, "Kokkos for MPI" ?

Comments/Thoughts/Suggestions/Snark:
jjellio@sandia.gov

Center for Computing Research