A Memory Consumption Benchmark for MPI Implementations

Samuel K. Gutiérrez
Computer, Computational, and Statistical Sciences Division
Los Alamos National Laboratory
samuel@lanl.gov

November 14, 2018

Abstract

This report describes the design, implementation, and usage of mpimemu, a memory consumption benchmark for implementations of the Message Passing Interface (MPI).

1 Introduction

Message passing libraries such as Open MPI [1] and MPICH [2] are examples of message-passing middleware. Like the *application drivers* they support, these libraries consume memory to maintain their internal state, which is primarily influenced by their software architecture (i.e., *how* they are implemented), runtime configuration (e.g., the underlying communication protocols used), and how they are driven with respect to job size (e.g., the size of `MPI_COMM_WORLD`) and communication workload (e.g., the communication pattern formed by the application driver).

Determining the amount of memory consumed by an MPI library (or implementation) is challenging and becoming increasingly important in the development, upkeep, and deployment of parallel programs in the high-performance computing (HPC) domain. Popular approaches for MPI library memory attribution can generally be categorized as library-specific instrumentation or benchmark-driven library analysis. An example of the former can be found in Cray's implementation of MPICH, where through environmental controls internal memory monitoring statistics are accessible via textual output (either to a terminal or a file). An example of the latter is *mpimemu*, the benchmark program described here.

2 History and Background

In 2011, mpimemu was written at Los Alamos National Laboratory (LANL) to help assess MPI library memory consumption as a function of job size, communication workload, and hardware/software architecture. mpimemu has since been included as a part of a broader suite of applications—*acceptance tests*—used to evaluate large HPC system installations.

2.1 Collecting Process/System Information

The *proc pseudo file system* (*procfs*) offers a convenient interface for obtaining information about and influencing the state of a running operating system (OS) kernel [3]. *procfs* provides user-space access to kernel-maintained state by exposing a file-based access semantics to the structure hierarchy it maintains (*directories* and *files*). Obtaining information about current OS state, including that of active processes, is accomplished by opening and parsing *files* located below *procfs*'s mount point (typically `/proc`). In

many cases, the content of these special files is generated dynamically to provide an updated view of the operating system's state.

Memory map monitoring collects information by interrogating specific *procfs* entries. This approach is appealing for a variety of reasons. First, it is relatively straightforward to implement. Second, when compared to heap profiling alone, it can provide a more holistic view into important features that ultimately impact a process's actual memory footprint, for example, the size and count of private/shared pages and their occupancy in RAM. Finally, it is language-agnostic and therefore readily applicable to any running process.

2.2 The Benchmark Program

mpimemu is an MPI program written in C with built-in memory map monitoring that works by sampling `/proc/self/status` (process level) and `/proc/meminfo` (node level), while optionally imposing a scalable communication workload on the system. The communication workload logically consists of the following parts:

1. *Small Data Allreduce Max*: A collective call wherein processes in `MPI_COMM_WORLD` exchange double-precision floating-point values via `MPI_Allreduce(MPI_MAX)`. The rationale behind including this particular operation in our synthetic communication workload is to emulate a common operation found in many HPC applications.

2. *Point-to-Point Exchange*: A series of point-to-point data exchanges between intra- and inter-node processes using `MPI_Sendrecv()`. Successive calls to this routine yield different data transmission totals, where sender-side totals cycle through sizes ranging from 1 B to 16 KiB. The rationale here is to exercise different communication protocols typically used in MPI implementations (e.g., *eager* and *rendezvous* protocols).

When enabled, communication and data collection are interleaved. That is, before each sample is collected the previously described data exchange operations are performed. Figure 1 shows the point-to-point communication patterns formed by mpimemu when given different runtime configurations. In this case, different numbers of processes per node (PPN) at 128 processes. For more details about mpimemu's communication workload, please consult `mmu_mpi_work()` in `mmu_mpi.c`. Runtime memory attribution in mpimemu is approximated by calculating usage deltas between samples collected during its execution and those collected before the initialization of the MPI library.

3 Obtaining the Benchmark Program

Source code distributions of mpimemu can be found at http://hpc.github.com/mpimemu. Pre-release source code can be found at https://github.com/hpc/mpimemu.

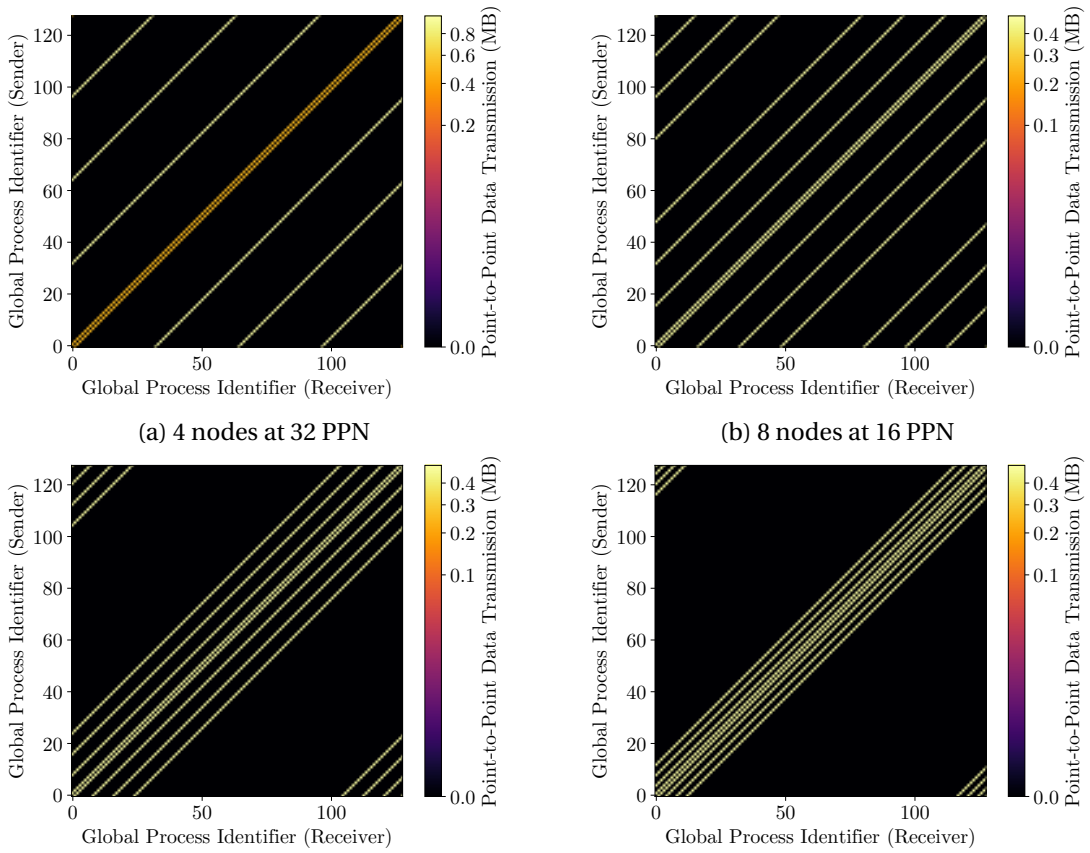
4 Building the Benchmark Program

First, make sure that `mpicc` or some other wrapper compiler with similar capabilities is in the `$PATH` (`mpicc` and `cc` are checked by default). To initiate a standard configure, invoke the following command:

```
./configure
```

To configure mpimemu with a different wrapper compiler, explicitly specify the wrapper. For example,

```
./configure CC=[NEWCC]
```



(a) 4 nodes at 32 PPN

(b) 8 nodes at 16 PPN

(c) 16 nodes at 8 PPN

(d) 32 nodes at 4 PPN

Figure 1: Visualization of the point-to-point communication structure formed by mpimemu when given different runtime configurations at 128 processes.

Once mpimemu's configuration process has completed successfully, build the source by invoking `make`. For example, one can configure, build, and install mpimemu with the following commands:

```
./configure --prefix=$HOME/local/mpimemu && make && make install
```

4.1 Installation Notes

The use of mpimemu does not require its installation (via `make install`). Rather, once built, mpimemu is usable from within its source directory.

5 Using the Benchmark Program

The preferred way to conduct a memory usage scaling study with mpimemu is to use a helper utility called `mpimemu-run`. `mpimemu-run` is used to run a succession of mpimemu instances at varying sizes. The following environment variables change the way `mpimemu-run` behaves.

5.1 Required Environment Variables

- MPIMEMU_MAX_PES: Specifies the maximum number of MPI processes to execute (i.e., job size) for a given experiment.
- MPIMEMU_RUN_CMD: Specifies the *run template* used to launch mpimemu jobs.
  - Example template specifications using mpirun:
    ```
    mpirun -n nnn aaa
    mpirun -n nnn -npernode NNN aaa
    ```
  - Example template specifications using aprun:
    ```
    aprun -n nnn aaa
    aprun -n nnn -N NNN aaa
    ```
  - Required template variables:
    nnn: Replaced with the total number processes to be launched. This value changes after each run and is determined by MPIMEMU_NUMPE_FUN(X). More on MPIMEMU_NUMPE_FUN below.
    aaa: Replaced with the mpimemu invocation string.
  - Optional template variables:
    NNN: Replaced with MPIMEMU_PPN. If MPIMEMU_PPN is not specified, a value of 1 will be used.
- MPIMEMU_NUMPE_FUN: Specifies the function that determines how nnn grows with respect to X. "X" must be provided in the string defining the function.
  - Example (bash-like):
    ```
    export MPIMEMU_START_INDEX=1
    export MPIMEMU_NUMPE_FUN="X + 1"
    export MPIMEMU_MAX_PES=4
    # Will run jobs of size 1, 2, 3, and 4.
    ```
  - Useful, accepted arithmetic operators:
    ```
    + Addition
    * Multiplication
    ** Exponentiation
    ```

5.2 Optional Environment Variables

- MPIMEMU_START_INDEX: Specifies the starting integer value of an increasing value, X. When set, X starts at the given value and is then incremented by 1 while MPIMEMU_NUMPE_FUN(X) $\leq$ MPIMEMU_MAX_PES. Note that a job size of MPIMEMU_MAX_PES will be included irrespective of the values yielded by MPIMEMU_NUMPE_FUN(X).
- MPIMEMU_PPN: Specifies the number of MPI processes per node.
- MPIMEMU_DATA_DIR_PREFIX: Specifies the base directory where mpimemu data are stored.
- MPIMEMU_BIN_PATH: Specifies the directory where mpimemu is located.
- MPIMEMU_SAMPS_PER_S: Integer value specifying the number of samples to collect every second. The default is 10 samples per second.

- `MPIMEMU_SAMP_DURATION`: Integer value specifying the sampling duration in seconds. The default is 10 seconds.
- `MPIMEMU_DISABLE_WORKLOAD`: If set, disables previously described synthetic MPI communication workload. See Section 2.2 for more details.

5.3 Running the Benchmark Program

Once the runtime environment is setup properly, add mpimemu's installation prefix to the `$PATH` or run from within its source distribution. For example (if in `$PATH`),

```
mpimemu-run
```

or from within mpimemu's source distribution

```
./src/mpimemu-run
```

When complete, a path to the generated data will be echoed to the terminal. For example,

```
data written to: /users/samuel/mpimemu-samuel-01032013
```

5.4 Generating CSV Files From Output Data

`mpimemu-mkstats` consolidates data generated by `mpimemu-run`. To generate comma-separated values (CSV) files, run the following command:

```
mpimemu-mkstats -i /path/to/data
```

For more options and information about `mpimemu-mkstats`, run `mpimemu-mkstats -h`.

5.5 Plotting CSV Data

`mpimemu-plot` can be used to generate node and process memory usage graphs from output generated by `mpimemu-mkstats`. `gnuplot` and `ps2pdf` are external programs that must be included in the `$PATH`, as they are used internally by `mpimemu-plot`.

5.6 Interpreting the Benchmark Data

To interpret the benchmark data, consider idle system memory usage (e.g., system image size). mpimemu provides a general sense of memory usage scaling characteristics, but it is important to note that not all memory consumed on a compute resource is attributable to the MPI library. Generally, it is a good idea to start scaling studies at 1 MPI process to get a general sense of *close-to-base* memory usage (i.e., base system usage). `mpimemu-plot` presents MPI memory usage as `MemUsed` − `PreInitMemUsed`. Please contact me if a better default memory usage metric exists.

5.7 Usage Notes

Please note that mpimemu requires all nodes to have the same number of processes. This implies that the process-per-node invariant must be maintained for all values of `MPIMEMU_NUMPE_FUN(X)`. Otherwise,

mpimemu will detect a violation of this requirement and will subsequently terminate the current job, resulting in an incomplete data set.

6 Acknowledgement

Work supported by the Advanced Simulation and Computing program of the U.S. Department of Energy's NNSA. Los Alamos National Laboratory is managed and operated by Los Alamos National Security, LLC (LANS), under contract number DE-AC52–06NA25396 for the Department of Energy's National Nuclear Security Administration (NNSA).

References

[1] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation". In. 11th European PVM/MPI Users' Group Meeting. Budapest, Hungary, September 2004.

[2] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard". In *Parallel Computing* 22(6) 1996, pages 789–828.

[3] *proc (5) Linux User's Manual.* December 2015.