A Scalable PMIx Database

Artem Y. Polyakov[1], Joshua Ladd[1], Elena Shipunova[2], Boris I. Karasev[1]

[1]Mellanox Technologies, [2]Intel, Inc.

Introduction

A Process Management Interface (PMI) is the component of the HPC software stack that is responsible for interaction between a Resource Manager and a parallel application.

In all PMI versions existing to date, the informational exchange between RM and application is organized in the form of a key-value database (KVDb) that has *Put*, *Get* operations and API-specific synchronization primitives.

The PMI Exascale (PMIx) (pseudo)standard [1] provides advanced capabilities to enable efficient bootstrapping of applications on emerging exascale systems.

This work focuses on the problem of scalable distribution of the job-level and application-specific data from PMIx server to PMIx client at the application side.

PMIx KVDb access specifics

- Shared memory technology significantly improves intra-node KVDb access latency [2] for PMI1/2.
- PMIx relaxes synchronization assumptions guaranteed by PMI1/2 (on-demand key fetch feature).
- An extension of the approach proposed in [2] with *lock-based KVDb access coordination* is required.

PMIx version and evaluation

- Considered PMIx version: **2.1**
- For PMIx *Get* performance estimation a *pmix_perf* microbenchmark from PMIx distribution was used.
- Processes were mapped by adjacent logical CPUs yielding gradual filling of a certain hardware resource before using the next one.
- Logical CPUs selection: *cores* for Intel system, *hardware threads (hw threads)* for IBM system.

PMIx *Get* fast-path algorithm

❶ *Perform a thread shift*: transfer of the control to PMIx service thread (ensures the thread safety)

❷ *Lock* KVDb for reading.

❸ Attempt to *fetch* the requested key from the shared memory.

❹ *Unlock* KVDb.

The curve *pmix/v2.1* on figures 1 and 2 represents the growth of PMIx *Get* operation latency on IBM and Intel systems.
On both systems latency grows significantly with number of PMIx clients.

PMIx *Get* fastpath optimizations

Profiling of PMIx *Get* showed that locking (steps 2 and 4, see PMIx *Get* fastpath) is the bottleneck.

However, we start with a set of code cleanup optimizations of obvious inefficiencies on step (3) in order to isolate and attribute subsequent improvements to an improved locking scheme.

- *First*, we eliminated thread shifting (step 1) from the fast path of the *Get* algorithm as the shared memory component does not access the global state of PMIx.
- *Second*, we removed unneeded memory allocations on the critical path replacing them with pre-allocated objects provided by PMIx *Get* caller.

The curve *fastp-opt* (figures 1 and 2) corresponds to PMIx version 2.1 extended with optimizations above.

Existing locking scheme limitations

PMIx utilizes Pthread Read/Write locks (RW-locks) to ensure that clients read accesses are consistent with the server-side updates (write access).

As demonstrated by the curves *pmix/2.1* and *fastp-opt*, this approach does not scale well with the number of application processes/PMIx clients (usually defined by available logical CPUs).

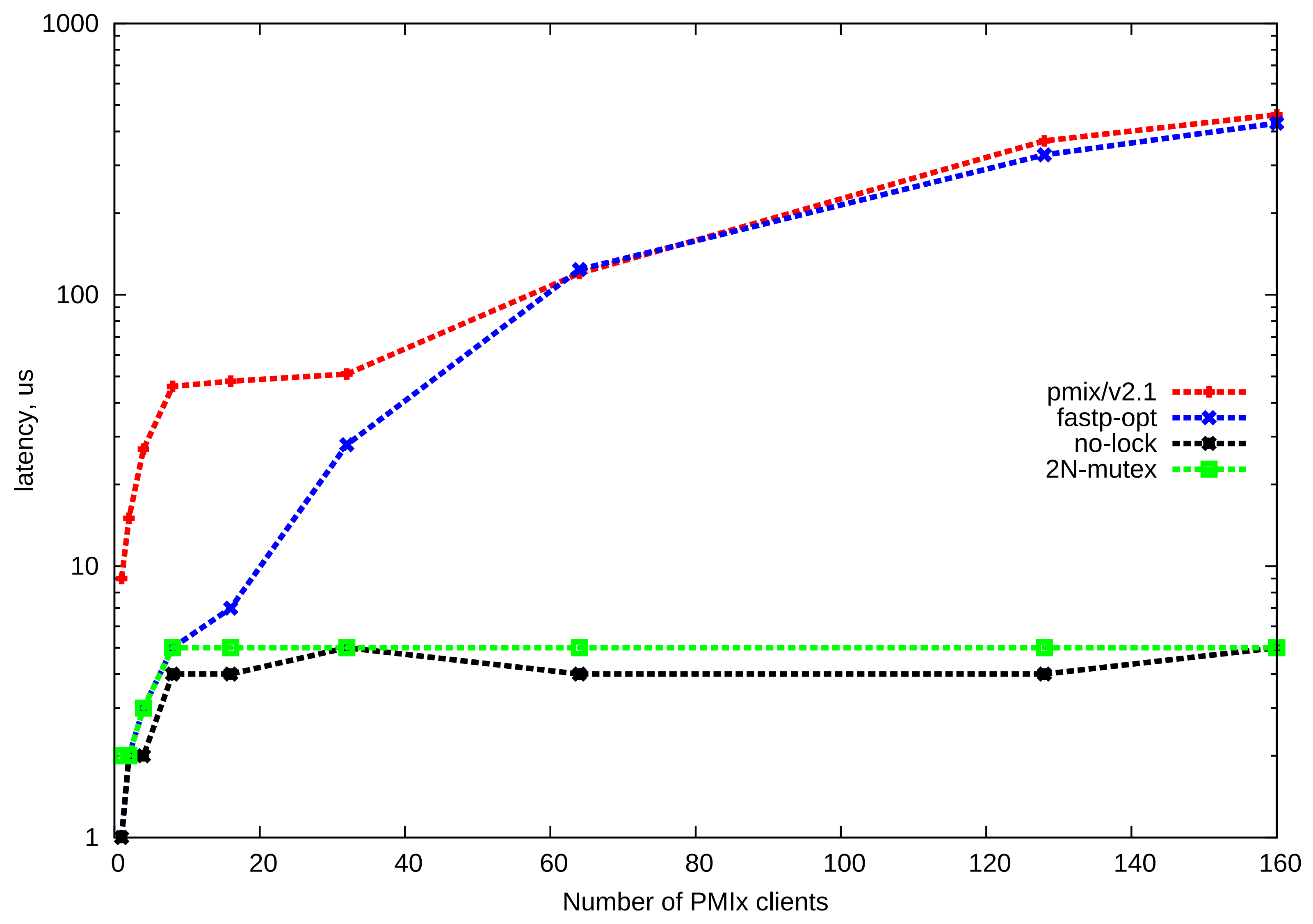


Figure 1: PMIx *Get* latency on IBM POWER8 system (2 sockets/20 cores/160 hw threads)

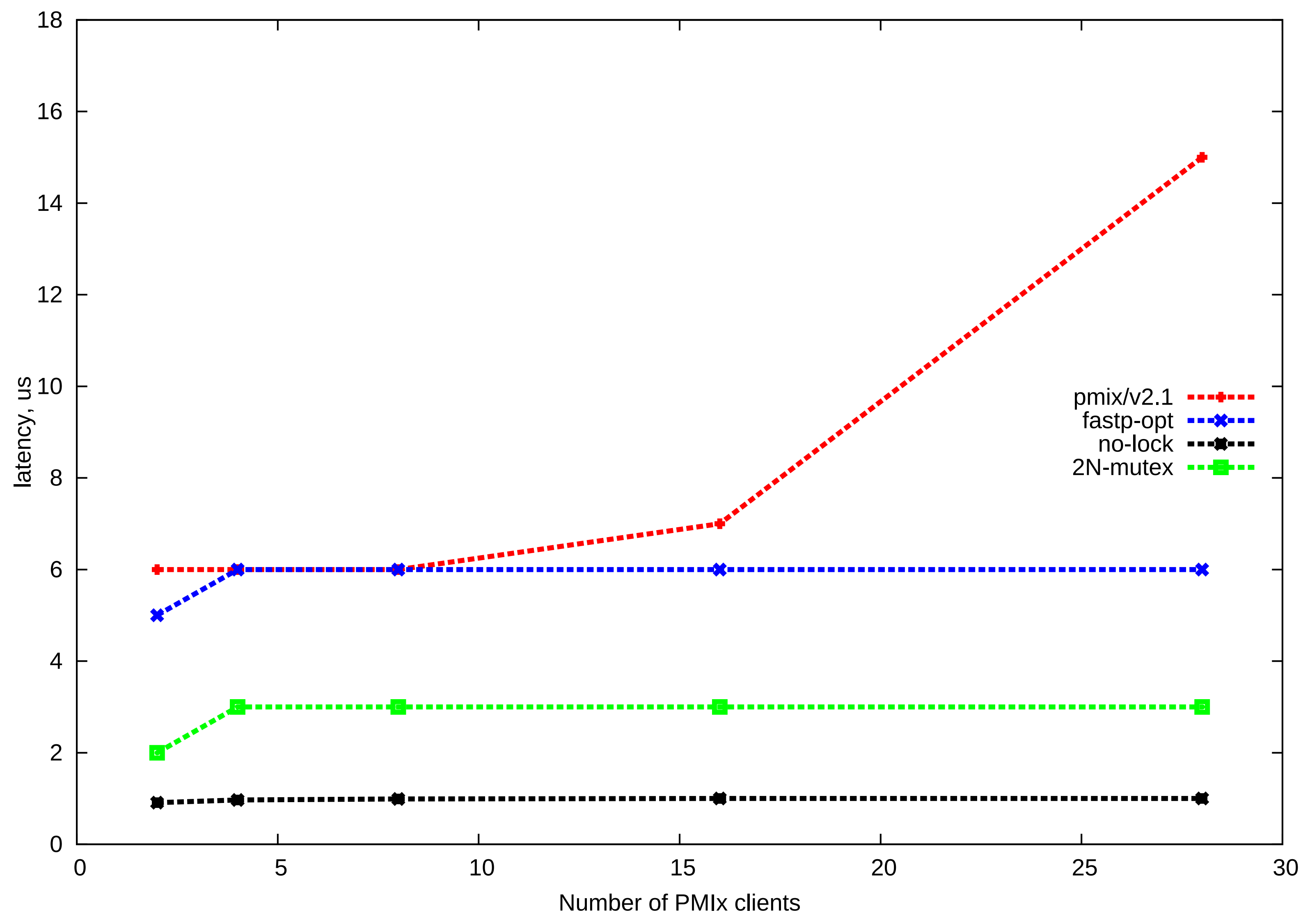


Figure 2: PMIx *Get* latency on Intel x86_64 Broadwell system (2 sockets, 28 cores)

PMIx database locking

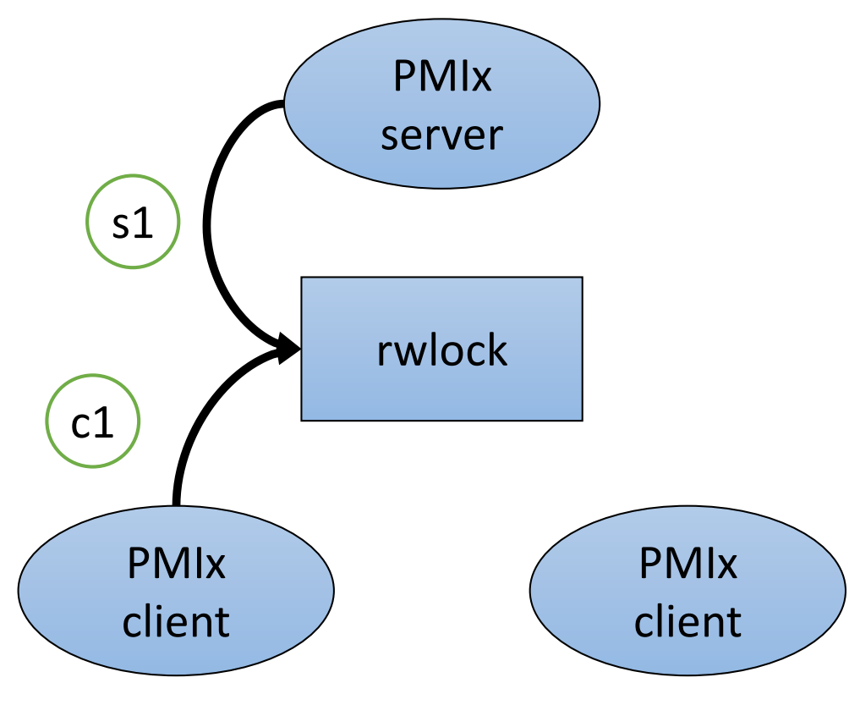
The problem of scalable RW-locks is well known [3].

However, PMIx database has several characteristics that distinguish it from the generic problem.

- KVDb has only one writer (PMIx server) thus no arbitration between multiple writers is required.
- Write locks are only present in PMIx *on-demand* mode where only a few keys expected to be exchanged.
- Readers are typically assigned on execution units (cores or hardware threads) while the writer does not have a dedicated hardware resource.
- Readers requesting the data are blocked waiting for the completion on the out-of-band channel.

Improved locking scheme

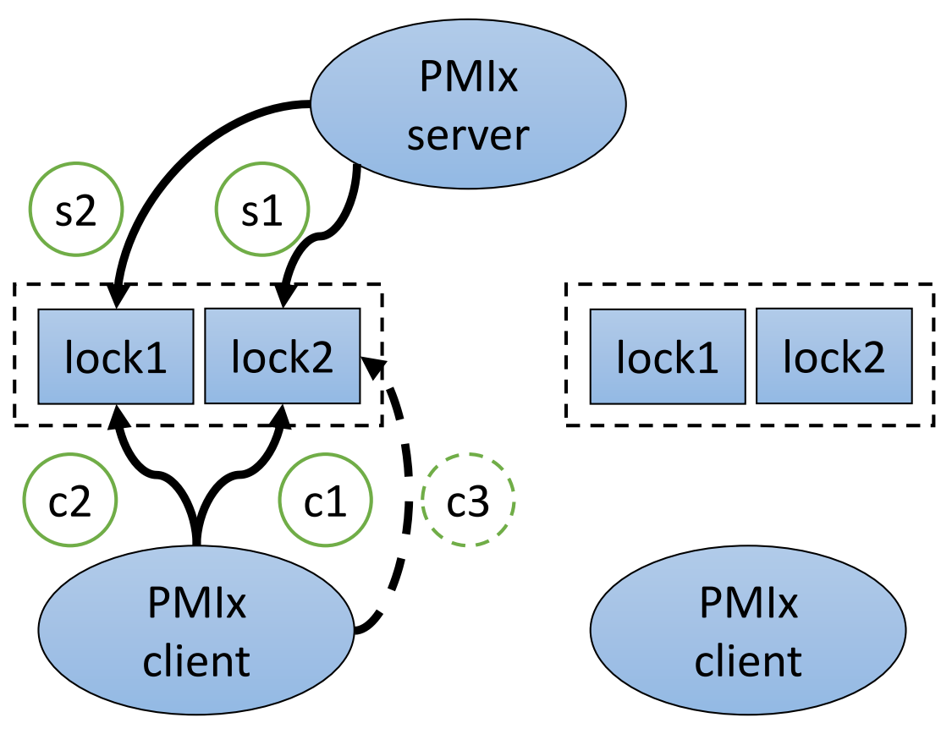
Based on these observations, we prioritize a reader scalability attribute and propose a *2N-lock* scheme (fig. 4) derived from the *static* approach [3]. The key difference of the *2N-lock* scheme is that it implements a writer-preference policy typical for PMIx scenario.



```
PMIx server lock procedure:
s1. lock_write(rwlock)

PMIx client lock procedure:
c1. lock_read(rwlock)
```

Figure 3: Existing PMIx KVDb locking scheme



```
PMIx server lock procedure:
// Get a signaling lock
for i in 1 ... cli_count do
s1. lock(cli[i].lock2)
// Get the main lock
for i in 1 ... cli_count do
s2. lock(cli[i].lock1)

PMIx i'th client lock procedure:
// Get the signaling lock
c1. lock(cli[i].lock2)
// Get the main lock
c2. lock(cli[i].lock1)
// Release the signaling lock
c3. unlock(cli[i].lock2)
```

Figure 4: Proposed PMIx KVDb locking scheme (*2N-mutex*)

Improved locking scheme(2)

The curve *2N-mutex* (fig. 1 and 2) demonstrates that on IBM system the performance of *2N-mutex* is close to a lockless case (curve *no-lock*). 2x on Intel system.

References

[1] Ralph Castain, David Solt, Joshua Hursey, Aurelien Bouteiller PMIx: Process Management for Exascale Environments. ACM, New York, pp. 14:1–14:10, 2017

[2] Chakraborty, Sourav and Subramoni, Hari and Perkins, Jonathan and Panda, Dhabaleswar K. SHMEMPMI – Shared Memory Based PMI for Improved Performance and Scalability. IEEE, New York, pp. 60–69, 2016

[3] Hsieh, W.C. and Weihl, W.E Scalable Reader-Writer LocksforParallel Systems. IEEE, New York, pp. 656–659, 1992

Contact information

- Artem Y. Polyakov, PhD
- Sr. Architect SW, Mellanox Technologies
- Email: artemp@mellanox.com