Towards Efficient Oversubscription: On the Cost and Benefit of Event-Based Communication in MPI

Jan Bierbaum
TU Dresden

Maksym Planeta
Barkhausen Institut

Hermann Härtig
TU Dresden

***Abstract*—Contemporary HPC systems use batch scheduling of compute jobs running on exclusively assigned hardware resources. During communication, polling for progress is the state of the art as it promises minimal latency. Previous work on oversubscription and event-based communication, i.e. vacating the CPU while waiting for communication to finish, shows that these techniques can improve the overall system utilisation and reduce the energy consumption. Despite these findings, neither of the two techniques is commonly used in HPC systems today. We believe that the current lack of detailed studies of the low-level effects of event-based communication, a key enabler of efficient oversubscription for classical MPI-based applications, is a major obstacle to a wider adoption.**

We demonstrate that the `sched_yield` system call, which is often used for oversubscription scenarios, is not best suited for this purpose on modern Linux systems. Furthermore, we incorporate event-based communication into Open MPI and evaluate the effects on latency and energy consumption using an MPI micro-benchmark. Our results indicate that event-base communication incurs significant latency overhead but also saves energy. Both effects grow with the imbalance of the application using MPI.

I. Introduction

The applications prevalent in HPC are compute-heavy and data-intensive. They are typically based on the *message passing interface* (MPI) [1], a ubiquitous standard for communication and synchronisation that is implemented by various libraries. This common standard allows for portability of applications between systems with very diverse networking hardware. HPC applications run large computations on distributed systems and are, therefore, very demanding when it comes to communication. They usually spawn one *rank* (an MPI term for process) per available CPU core and, when waiting for communication operations to complete, rely on busy waiting or *polling* for minimal end-to-end latency.

Polling greedily occupies all allocated resources, even when no computational progress is made. In contrast, *event-based* communication has a rank relinquish its CPU until the runtime system or kernel signals the completion of one or more communication operations. This strategy lowers the CPU utilisation, which promises significant advantages with respect to energy consumption. Event-based communication also lends itself particularly well to oversubscription, in which other applications utilise the vacant CPU. Despite these advantages, there is currently little support for event-based communication in popular MPI libraries. Instead, when running

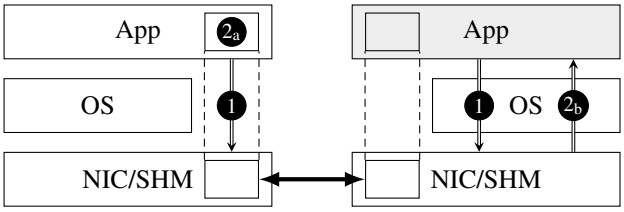


Figure 1: Comparison of Communication Modes. An application first sets up a new (send or receive) operation ❶ via the network interface card (NIC) or shared memory (SHM). In polling mode, shown on the left side, the application subsequently polls its memory to check for the operation's completion ❷a. In event-based mode, shown on the right side, the application sleeps instead. Once the communication finishes, the operating system wakes up the application ❷b.

in oversubscribed environments, MPI libraries give up the CPU via the `sched_yield` system call [2]. But as we show, yielding cannot replace a faithful implementation of event-based communication in this context.

We find that event-based communication is underrated, understudied, and lacks wider adoption as a consequence. In this paper, we present our modifications of Open MPI to support event-based communication. We then study the mode's effects at the lowest possible level using Libra, a ping-pong micro-benchmark we designed for this exact purpose. Furthermore, we show the problems of using `sched_yield` as a substitute for event-based communication in oversubscribed environments.

Following an exploration of relevant concepts (Section II) and related work (Section III), we introduce our modification of Open MPI (Section V) and discuss `sched_yield` ()Section VI). Section VI presents the Libra micro-benchmark as well as our measurement setup and results before we summarise our findings and conclude in Section VII.

II. Background

The batch processing of applications, used by virtually all current HPC systems, assigns resources exclusively. An application waits until the system's job scheduler allocates a dedicated partition of the available hardware resources to it. In typical HPC setups, the granularity of these partitions is full compute nodes, not individual CPUs, so different applications do not share any resources except for the network. The hardware and the runtime system are deemed fully deterministic. An

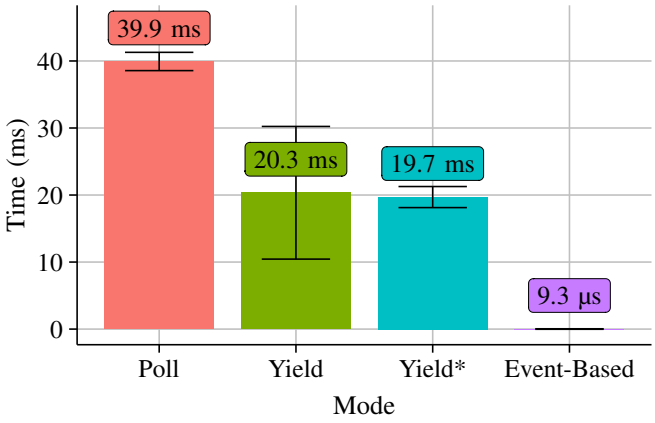


Figure 2: Oversubscription of the LIBRA benchmark using different communication modes. The graph shows the average latency and standard deviation of one ping-pong operation. Both ranks share a CPU and we simulate an imbalance of 50 µs. Section VI provides more details on the benchmark and measurement setup. "Yield" refers to the standard implementation of `sched_yield` in Linux, whereas "Yield*" is a patched version (Section IV).

application with full control over its partition can, therefore, make an informed decision on how to use its resources most efficiently. Consequently, MPI-based applications use one rank per available CPU core and pin it to that core to effectively circumvent the operating system scheduler. To minimise end-to-end communication latency, ranks *poll* in a busy loop to check their send or receive operations for completion; shown on the left side of Figure 1. However, this strategy leads to inefficiencies when applications cannot fully utilise their assigned resources due to load imbalances or hardware performance variations [3].

During its runtime, each rank alternates between phases of independent parallel computation and phases of communication and synchronisation [4]. A *straggler*, a slow rank entering its communication phase late, may not only delay the whole computation, but force other ranks to also waste compute cycles and energy while waiting for the slow one. Countermeasures include dynamic load balancing and overlapping of computation and communication phases.

These strategies, however, are cumbersome and error-prone for application developers who are often domain experts but not necessarily MPI experts. The code's complexity increases, making it harder to read and maintain [5]. With the expected increases in the size of HPC clusters and other factors like heterogeneity and hardware performance variation, the aforementioned problems will become only more pronounced. Sometimes, the appearance of stragglers is totally out of control of application programmers. For example, even the CPU clock rate may depend on the instructions executed and manufacturing-induced variations [3, 6, 7]. Therefore, stragglers should be addressed at the operating system and runtime level as well.

A. Oversubscription

Oversubscription is a technique that can improve the system utilisation and throughput. Multiple ranks (potentially from multiple applications) run on the same CPU cores. This approach allows the runtime or operating system to take care of the issues discussed previously, for example by changing how many and which ranks share a core. But efficient oversubscription is also a useful tool for developing HPC applications on ordinary desktop machines with relatively few cores.

However, naive oversubscription based on polling communication will achieve the exact opposite as demonstrated in Figure 2. When multiple ranks share a core, CPU cycles used to poll for the completion of communication operation might better be used by another rank's computation. As polling is so disastrous to performance in oversubscribed scenarios, the `sched_yield` system call is used instead; see Section III. Open MPI, for example, even applies it automatically when the library detects oversubscription. `sched_yield` is expected to trigger a de-scheduling of the currently running rank, allowing another rank to run on the CPU. This is not valid anymore in modern versions of Linux as we will further discuss in Section IV. Figure 2 also shows the effects of yielding, both for the default implementation of Linux and our modified version (see Section IV).

B. Event-based Communication

Event-based communication, shown on the right side of Figure 1, offers an alternative to polling for completion. A rank will relinquish its CPU until the runtime system or kernel signals the completion of one or more communication operations. This allows either another rank to make progress, which is especially compelling for oversubscription, or the CPU to idle and potentially enter energy-saving states. While single application performance has been the dominating quality metric of HPC systems for a long time, energy is gaining importance; a trend that is accelerated by the advent of Exascale systems. Frontier, the first such system, consumes as much as 21 MW of electrical power [8] and the peak power of Aurora, an upcoming Exascale machine, is expected to be up to 60 MW [9].

Another convenient side effect of event-based communication leaving the CPU idle is that traditional system monitoring tools like `top` allow an easily accessible way to judge the efficiency of the parallel application. Whereas polling appears the same as useful computation, idle phases show up in these tools. However, event-based communication is not widely supported by MPI libraries. To the best of our knowledge, only MVAPICH [10] offers such support, limited to its InfiniBand backend.

C. Open MPI

Open MPI [11] is one of the most commonly used MPI libraries. It is free software and of particular interest to us due to its modular and well-documented codebase. For our experiments, we enabled event-based communication in Open MPI as detailed in Section V.

Unified Communication X (UCX) [12] is an open-source communication framework used as the default communication backend for InfiniBand in Open MPI. UCX unifies different vendor- and device-specific drivers on a lower level than MPI and provides only few, essential messaging primitives. The library comes with support for event-based communication for various backends, notably including InfiniBand and shared memory, which are often found in HPC settings. Even though each backend requires a distinct implementation — interrupts generated by the NIC, shared memory ultimately relies on Unix domain sockets [13] — all mechanisms converge in a file descriptor that becomes readable once a communication operation finishes. UCX uses this file descriptor in a `poll` system call [14] to implement event-based communication. For reason discussed in Section V, we extended the existing implementation to allow for a timeout. Even without any completion, `poll` will return after this pre-specified time and thus not suspend execution indefinitely.

III. Related Work

Utrera *et al.* use oversubscription to provide MPI-based applications with "virtual malleability" [15]. Normally, the user decides the number of processes of an MPI application when launching the programme. Any changes at runtime, if done at all, need to be triggered by the application. Malleable applications, on the other hand, can adapt to runtime changes in the resources available to them, providing the system with more flexibility. Virtual malleability makes MPI applications malleable transparently. The system uses oversubscription should there be fewer cores available than application ranks.

Building on this work, Utrera *et al.* propose a job scheduler that maximises resource utilisation and improves overall system performance by allowing jobs to adapt to variations in the load through virtual malleability [16]. Considering only single applications in a heterogeneous environment, Utrera *et al.* also applied oversubscription to "pack" MPI applications [17]. The performance difference between more and less powerful core naturally causes stragglers. Running multiple processes oversubscribed on a the more powerful cores, frees up some of these cores without significantly affecting the application's wallclock runtime.

"Tangram" [18] fills partially idles nodes by co-locating applications and selective use of oversubscription. Sharing nodes between CPU-intensive, memory-intensive, and I/O-intensive applications allows the system to reduce the overall makespan as well as the turnaround time for individual applications.

In all of the aforementioned works, the authors either use `sched_yield` for efficient oversubscription or do not discuss the specific mechanism at all. In the latter case, `sched_yield` is most likely applied implicitly by the MPI library which detects the oversubscription situation automatically. Other alternatives, especially event-based communications, have not been explored at this point.

```
1  end = now() + 30s
2  counter = 0
3
4  while now() < end
5      counter += 1
6      if DO_YIELD
7          sched_yield()
8      else
9          getppid()
10 print(counter)
```

Figure 3: Micro benchmark to showcase the effects of `sched_yield`.

[19] employs oversubscription for the in-situ analysis of a running simulation. The simulation uses a hybrid approach with MPI for its inter-node communication and OpenMP for node-local parallelisation. During serial phases of the simulation, determined via online monitoring and prediction, the analysis application utilises the now idle cores. The authors use `SIGSTOP` to suspend the analytics whenever the main simulation runs fully parallel. This way they avoid any performance degradation due to ranks competing for a CPU.

There are also oversubscription papers which consider only simultaneous multithreading as a kind of hardware-level oversubscription [20, 21]. Whereas this approach avoids the problem of efficient software-level oversubscription, it is also less flexible then the aforementioned schemes.

Venkatesh *et al.* demonstrated with "Energy-Aware MPI" that event-based communication provides significant advantages in terms of energy consumption [22] as it allows the CPU to idle and potentially enter energy-saving modes. The system uses no oversubscription and extends the event-based communication in MVAPICH to trigger based on expected communication latencies. The user can chose a maximally acceptable performance degradation in trade for reduced energy consumption.

"Adagio" [23] saves energy for classical MPI applications that rely on polling. Using predicted computing times, the runtime system reduced the clock rate of processors (dynamic voltage and frequency) that run "fast" ranks. This slows all ranks down to the speed of stragglers and saves energy dues to the reduced clock rate.

Other approaches incorporate higher-level abstractions [24–26], extend MPI [27, 28] or employ (user-level) threads instead of processes to represent MPI ranks [29–31]. This allows for lower overhead and finer control during oversubscription. However, non-trivial applications usually require manual adoption to work with or benefit from these runtimes.

IV. The `sched_yield` System Call

Polling is detrimental to performance in oversubscribed scenarios. Due to the lack of event-based communication in most available MPI libraries, they `sched_yield` system call is often used to trigger a de-scheduling of the current rank,

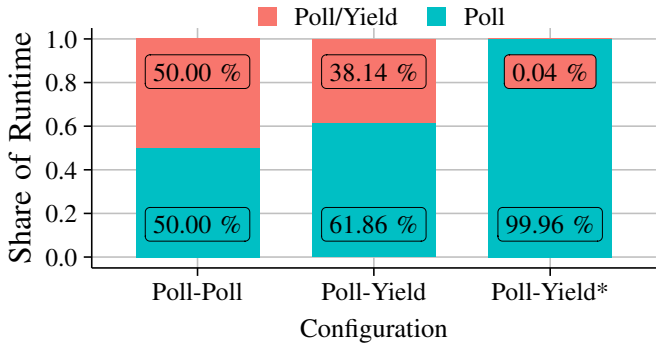


Figure 4: Effects of the `sched_yield` system call on the runtime allocated to two processes sharing a CPU. Contrary to common expectations, a process issuing the system call does not necessarily vacate its CPU. The effect is significantly more pronounced with the patched version of CFS ("Yield*").

allowing another one to run on the CPU. MPI libraries typically even apply it automatically when they detect oversubscription. In modern versions of Linux, however, the actual behaviour of `sched_yield` is not so straightforward.

A. Scheduling in Linux

The *Completely Fair Scheduler* (CFS), Linux' default scheduler, is designed to give each running process[1] a fair share of the available CPU time. To do so, CFS maintains a record of the *virtual runtime* of each process and bases its scheduling decisions on this value. Whenever a scheduling decision is due, CFS will select the process with the currently lowest virtual runtime as it did not yet have its fair share of the available computation time. The virtual runtime is proportional to the time a process occupied any CPU, weighted by the process's scheduling priority; a higher priority making the virtual time pass slower. A rank's priority can be influenced by the user but CFS also adjusts it automatically: compute-intensive ranks get demoted to provide a fairer distribution of available compute time and better interactive in end user systems. For traditional HPC applications, CFS has hardly any influence. Each rank is pinned to a dedicated CPU which runs nothing else. So priorities and even CFS as a whole do not play any signifiant role.

This situation changes drastically as soon as oversubscription comes into play: Now, multiple ranks compete for a CPU and get scheduled by CFS. As mentioned before, MPI libraries or runtimes (see Section III) use the `sched_yield` system call to influence the work of CFS, making it de-schedule the running rank, which polls for some communication operation to complete, in favour of another. However, modern implementations of CFS strive to uphold their fairness property even in the presence of `sched_yield`. The scheduler will tenaciously select the process with the currently lowest virtual runtime. When CPU pinning is involved, only processes allowed to run the CPU are eligible. Polling for the completion of communication requires the rank only to check a few memory locations for updates before it yields the CPU. With high probability, the polling rank still has the lowest virtual runtime and continues to run. So, in contrast to the programmer's intention, `sched_yield` has little to no effect in these scenarios.[2]

[1]CFS, being a Linux kernel scheduler, is concerned with processes or threads in general. For the purpose of this paper, we consider the terms "process" and "rank" equivalent and use whichever is more appropriate in the given context.

B. Demonstration of the Effects

We patched Linux to reinstate the (now deprecated) option to enforce `sched_yield` even within CFS. This patch closely resembles the implementation of `sched_compat_yield` that was available in kernel versions prior to 3.0[3] with small changes to work with current kernels. Whenever a rank invokes `sched_yield`, the patched CFS will look up the currently largest virtual runtime among all ranks assigned to the same CPU. The calling thread's virtual runtime is then set to this maximum + 1 to effectively "move it to the end of the queue". An entry in the `sysfs` pseudo file system [32] allows the user to switch between the regular implementation and the modified one at runtime.

To demonstrate the vastly different effect, consider the micro-benchmark shown in Figure 3. The benchmark runs for a fixed time of 30 seconds, continuously incrementing a counter variable in a tight loop. It also issues a system call in the loop; either `sched_yield` or `getppid`, depending a command line argument. This approach makes sure that the inherent overhead of a system call has no effect on the result. At the end, the benchmarks outputs the final value of its counter variable which serves as a measure of the compute time the benchmark received.

We run two instances of this benchmark in parallel on a single core, a setup similar to oversubscription. One of the two instances always has `DO_YIELD` set to `false`, so it does not call `sched_yield`. We calculate the share of runtime of instance X as `counter_X / (counter_1 + counter_2)`. Figure 4 shows the results for three different scenarios: In the baseline measurement, the second instance of the benchmark also uses `DO_YIELD = false`. As expected, both instances got equal shares of CPU time. In the remaining scenarios the second instance has `DO_YIELD` set to true. They differ only in the implementation of `sched_yield` being used and the effect of the "more aggressive" patched version is clearly visible.

To be able to use a modified Linux kernel, we had to run the benchmark on a test cluster in our lab. However, repeating the first two scenarios on the "Taurus" cluster bore similar results. We conclude that the Linux kernels shipped by HPC vendors are not different from the vanilla kernel in this particular aspect. Details on both systems are available in Section VI.

[2]Note, that according to documentation [2], the effects of `sched_yield` are unspecified for CFS. The stated effect is based on the actual implementation within the Linux kernel.

[3]Git commit `1799e35d` in the official Linux repository.

```
1  progress_loop()
2      static num_tries = 0
3
4      events = ucx_progress()
5      if events > 0
6          goto reset_tries
7
8      num_tries += 1
9      if TRIES_POLL == INFINITY or
           num_tries < TRIES_POLL
10         return
11
12     if TRIES_YIELD == INFINITY or
           num_tries < (TRIES_POLL + TRIES_YIELD)
13         sched_yield()
14         return
15
16     events = ucx_wait(TIMEOUT)
17     if events > 0
18 reset_tries:
19         num_tries = 0
```

Figure 5: Simplified progress loop of Open MPI with modifications for adaptive waiting.

V. Event-Based Communication in Open MPI

Open MPI advances any kind of communication in its *progress loop*. This regularly executing code queries all available communication backends for progress and triggers further actions when completions occur. During polling, the progress loop is running continuously. When Open MPI detects that it runs in an oversubscribed environment (or when manually instructed to do so), the progress loop incorporates a call to `sched_yield`. The intention is for the system scheduler to give the CPU to another process. Section IV discusses how this expectation is not necessarily met by modern versions of the Linux kernel.

We modified the existing progress loop to allow for *adaptive waiting*. In contrast to the readily available implementation of event-based communication in MVAPICH, ours works not only for InfiniBand but, in principle, for arbitrary backends. Currently, only UCX provides the required features for InfiniBand and shared memory. Furthermore, adaptive waiting allows to switch from tight polling to yielding the CPU to event-based communication at runtime. This scheme is inspired by similar approaches in the implementation of locks.

Figure 5 shows pseudo code for our implementation. In the original implementation, all available communication backends are progressed individually. That includes processing of outstanding messages and checking for completed communication operations. We concentrate on UCX as the main backend, so line 4 refers only to that. Note, that UCX supports shared memory communication, too, so adaptive waiting also works in the common scenario where InfiniBand and shared memory are in use at the same time.

If there are any completions, the number of unsuccessful tries is reset (line 6). Three user-controlled parameters further steer the communication strategy: `TRIES_POLL` specifies how often the progress loop is to run in a tight polling mode. The default value of `INFINITY` indicates to use this mode exclusively; the standard behaviour of Open MPI. In the actual implementation, we also use compiler hints to optimise for this scenario so as to not affect the performance of traditional MPI applications. As long as as we remain in polling mode, the progress loop returns (line 10) and higher layers of Open MPI will re-executed the function almost immediately.

After `TRIES_POLL` repetitions of the progress loop failed to see any communication operation complete, we call `sched_yield` for the following `TRIES_YIELD` attempts (line 13). The next run of the loop will, again, check all backends for completions. Note, that both aforementioned parameters can also be set to 0 to directly activate the last, the event-based mode. In this mode, the progress loop instructs UCX to suspend itself (see Section II-C for details) until either any communication operation finishes or the `TIMEOUT` elapses (line 16).

The timeout turned out to be necessary as Open MPI does expects its progress loop to finish fast and run repeatedly. Progressing the communication backends (line 4) also triggers processing of yet unsent messages. Unbounded waiting can therefore lead to deadlocks in some cases, especially in collective MPI operations or when other backends are used in parallel with UCX. Until we solve these issues, the timeout serves as a fallback.

VI. Evaluation

As demonstrated in II-A efficient oversubscription of classical MPI applications relies on event-based communication. To properly judge the cost and benefit of this communication mode, we devised the Libra micro-benchmark and used it to run extensive experiments.

A. Libra *Micro-Benchmark*

The main effects of event-based communication will show in the latency of MPI operations and their energy consumption. For our low-level study, we concentrate on point-to-point messages as collective operations are typically built on to of these primitives. One of the main use cases of oversubscription is to better cope with stragglers, i.e. ranks that are slower than their communication partner. Therefore, we want to be able to emulate their effect in the communication.

Initially, we considered using the point-to-point latency benchmark from the OSU benchmark suite [33]. However, this benchmark reports only average latencies and offers no way to emulate stragglers. Using the benchmark as an example, we developed Libra. Figure 6 shows a pseudo code version of Libra's core implementation: Its two ranks first synchronise and then exchange MPI messages in a ping-pong fashion for a configurable number of times. In one direction, Libra sends

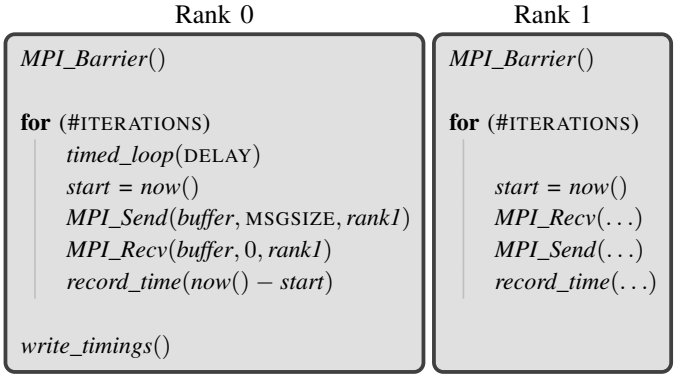


Figure 6: LIBRA MPI Micro-Benchmark

a message of configurable size, whereas the reply is empty to maintain better control over the amount of transferred data. To avoid any uncontrolled imbalance between the two ranks, both time each individual ping-pong operation. The time source for `now()` is `std::chrono::steady_clock`. In the end, rank 0 writes these values to a file for further analysis. This approach of recording the end-to-end ping-pong latency in one rank avoids any issues with unsynchronised clocks in the two ranks.

To emulate a straggler rank, we introduce a time-based loop in rank 0 which continuously polls the current time until the user-specified `DELAY` elapsed. This polling is equivalent to compute operations of an actual MPI-based application and is not part of the measured communication latency. The delay loop does, however, contribute to the overall energy consumption as we measure energy externally for a full run of the benchmark, not for individual ping-pongs.

B. Measurement Setup

For measurements shown in previous sections of this paper, we used a local test machine with a 4-core Intel Core i7-4790 CPU with Hyper-Threading enabled (but unused), 32 GiB RAM, and a Mellanox ConnectX-3 (MT27500) network adapter (also not used). In contrast to an HPC cluster, we were able to run a modified Linux kernel (version 5.12.1) on this machine.

All of the following experiments, we ran on the High Performance Computing and Storage Complex (HRSK-II or "Taurus")[4] at TU Dresden. Each node of this system has two 12-core Intel Xeon E5-2680 v3 CPUs with Hyper-Threading disabled, 64 GiB RAM, and a Mellanox Connect-IB (MT27600) network adapter. The system is equipped with "High Definition Energy Efficiency Monitoring" (HDEEM) [34], power instrumentation that allows to record energy consumption data for individual nodes and their CPUs. The system runs a standard Red Hat Enterprise Linux kernel (version 3.10.0-1127.19.1.el7.x86_64) provided by Bull.

For all benchmarks we used an exclusively allocated single node and pinned each rank to one of the CPUs. Not only does this method allow HDEEM to measure the energy for each rank separately, it also enables the CPUs to make full use of their energy saving modes. Using a single node avoids any external influences from other users. In particular that means, that measurements on the InfiniBand backend use the InfiniBand network adapter for communication but not the network. We deem this a valid approach as event-based communication has no influence on the propagation of messages within the network.

We used Open MPI 4.1.1 with UCX 1.10.1, both patched as described in Section V, together with our LIBRA micro-benchmark. Rank 0, the "measuring side" of the benchmark, always runs with polling-based communication to ensure minimal latency. The "measured side", rank 1, runs either with polling, (unmodified) yielding, or event-based. LIBRA ran with message sizes from 1 B up to 8 MiB and sender delays ranging from 0 µs to 1 ms. To keep the benchmarking time manageable, we selected the number of iterations according to message size and sender delay: 100 000 iterations for messages smaller than 1 KiB and delays shorter then 50 µs, then progressively fewer down to 100 iterations.

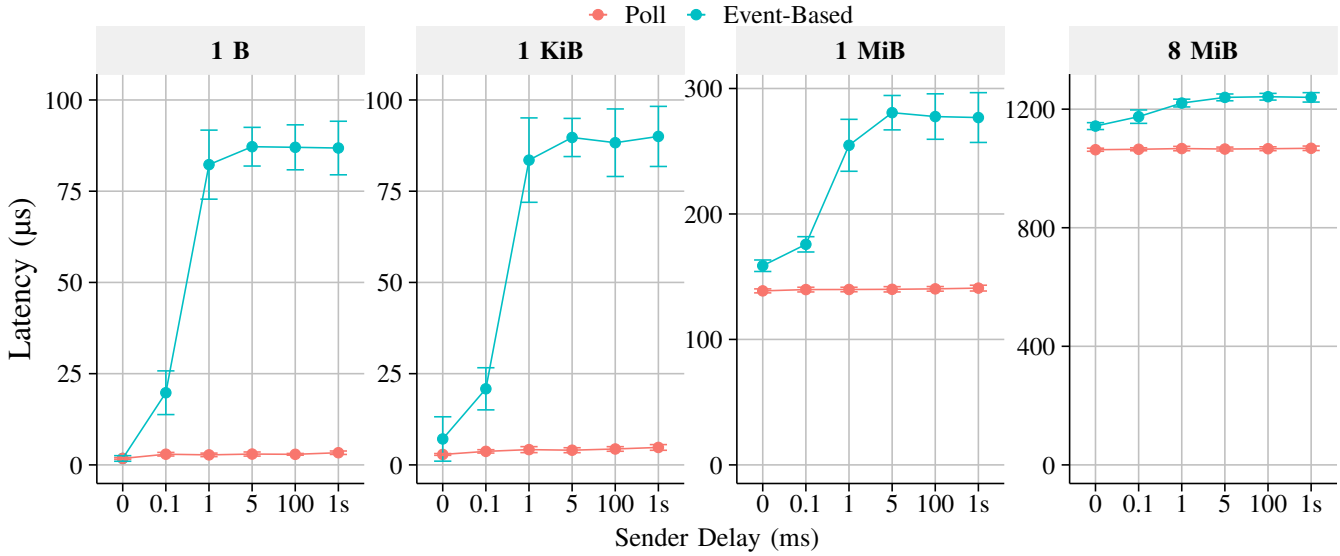
Different modes and backends have an effect on the communication. We consider the InfiniBand and the shared memory backend, to cover the typical communication paths for HPC applications: shared memory for ranks on the same node and InfiniBand for remote nodes. To study the properties of all available communication modes, we measured the times for polling (`TRIES_POLL=INFINITY`), yielding (`TRIES_POLL=0`, `TRIES_YIELD=INFINITY`), and event-based communication (`TRIES_POLL=0`, `TRIES_YIELD=0`, `TIMEOUT=100`). Different timeout values for event-based communication had no measurable effect so we omit them for brevity.

C. Results

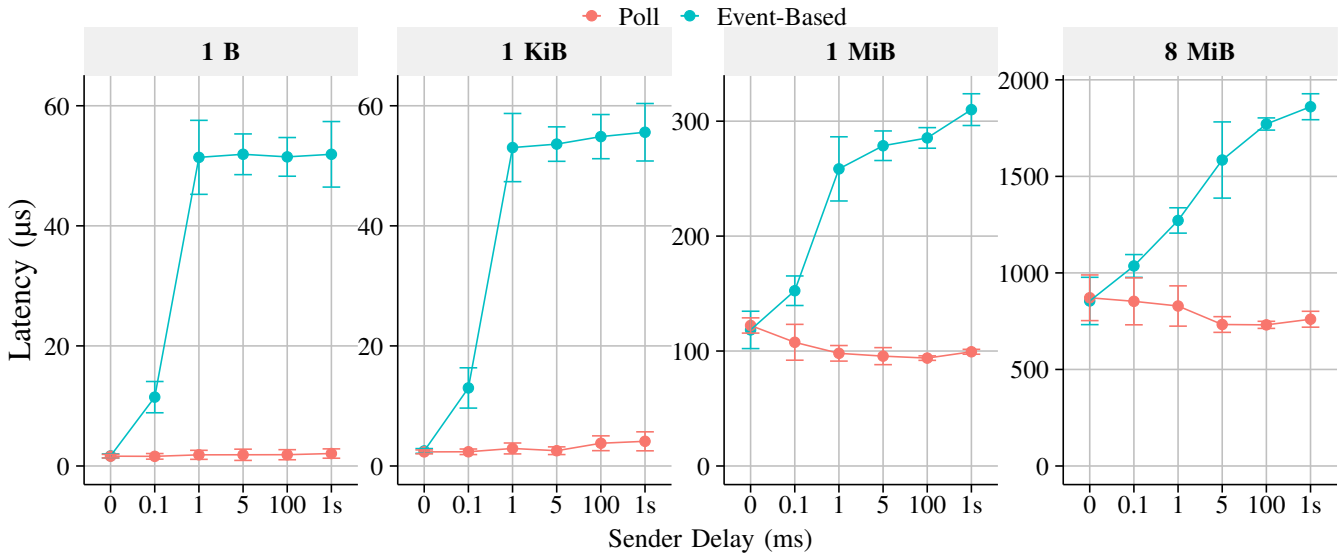
As there are no other processes sharing the CPU with the benchmark, we observe yielding to effectively become polling. The results for these two modes lie within a 5% range of each other. We therefore omit yielding from the graphs and further discussion.

a) Latency: Figure 7 depicts the average latency of a ping-pong operation across different message sizes and sender delays. Without any sender delay, i.e. for perfectly synchronised applications, there are only small differences between polling and event-based communication. This is because the first test for completion already succeeds and no waiting occurs. For higher delays we see an increased variance as well as a significant absolute overhead for event-based communication that levels out with increasing delays. For small messages this overhead stabilises at ≈ 90 µs for InfiniBandand ≈ 50 µs for shared memory. Considering the already large delay at this point, the additional overhead added by event-based communication is negligible. For large messages, the data transfer itself consumes more time which reflects in the latency; it increases for polling as well. As a consequence, the relative latency overhead of event-based communication gets lower, better amortising its cost.

[4]https://tu-dresden.de/zih/hochleistungsrechnen



(a) InfiniBand



(b) Shared Memory

Figure 7: Average latency of a ping-pong operation in the LIBRA micro-benchmark. Whiskers show the standard deviation. The latency increases with larger messages as the data transfer itself takes longer. As an result, the relative overhead of event-based communication becomes smaller with increasing message size.

We did not yet investigate the sources for the observed overhead in detail. But the longer the delay, the more likely rank 1 will actually wait and enter a sleeping state. Setting up the waiting mechanism, suspending the rank, and resuming it later on are likely the main source of the observed overheads but we also consider the power-saving mechanism of modern CPU to play an important role.

b) Energy: The assumption of power-saving mechanism having significant influence is strengthened by the energy consumption we observe in our experiments, shown in Figure 8. We omit results for very short sender delays as the method of measuring the energy consumption from outside the benchmark turned out to produce noisy results in these cases. Due to the short benchmark runtime the measurement captured too many unrelated external effects.

Considering only the CPU that runs rank 1 (Figure 8a), event-based communication is always beneficial using InfiniBand. Longer delays, emulating more imbalanced applications, strengthen this effect as the CPU remains idle for longer whereas in polling mode it runs a tight loop. In absolute numbers, the energy required for a small message grows by multiple orders of magnitude from $\approx$800 mJ to $>$30 J. Similar

to longer delays, larger messages lead to longer overall runtimes and, therefore, higher absolute energy consumption. Only for shared memory the message size has a noticeable effect on the relation between polling and event-based mode. This effect vanishes with longer delays. At this point, the causes are unknown and require further investigation.

With the overall system energy consumption (Figure 8b), the overarching trends are the same but the savings are less pronounced. This energy domain includes the second CPU running rank 0 as well as the power supply unit, local mass storage, the network adapter, and other peripherals. Compared to just the CPU running rank 1, the absolute energy consumed is larger by a factor of 5. However, only that CPU becomes idle in event-based mode and can save energy.

VII. Conclusion & Outlook

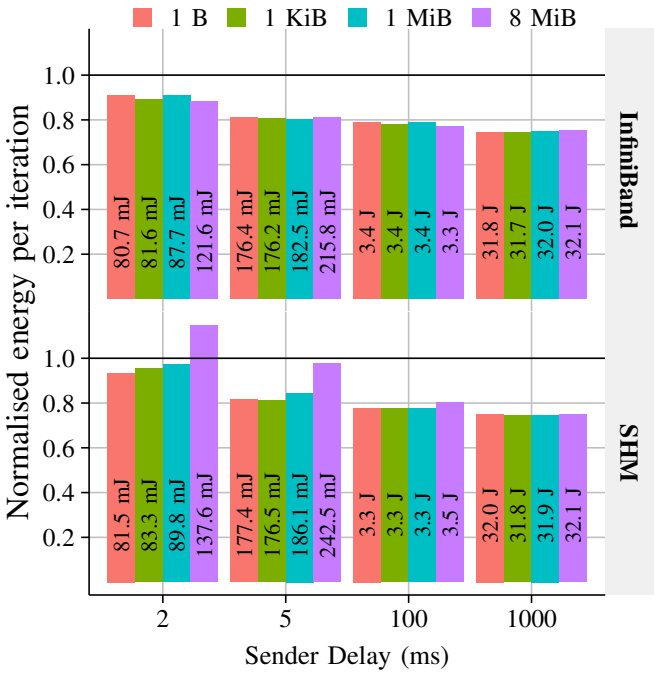
We have demonstrated event-based communication are a promising basis for oversubscription in HPC environments by implementing and evaluating this communication mode in Open MPI, a popular HPC communication library. For the evaluation, we developed Libra, an MPI micro-benchmark that can emulate imbalanced applications and measure the latency and energy consumption of low-level communication operations precisely. Furthermore, we showed the futility of using the `sched_yield` system call of modern Linux kernels for efficient oversubscription. All software patches discussed in this paper and the full benchmarking data are available online at https://zenodo.org/record/7198966.

Our results show that event-based communication incurs a latency overhead that is particularly significant for small messages and well-balanced applications. In these scenarios the traditional polling approach is better suited. For large messages, however, the overhead is less pronounced and especially for imbalanced applications event-based communication allows the CPU to save energy. One possible cause for both, higher energy savings and higher latency overhead, is that the CPU needs more time to wake from deeper sleep states.

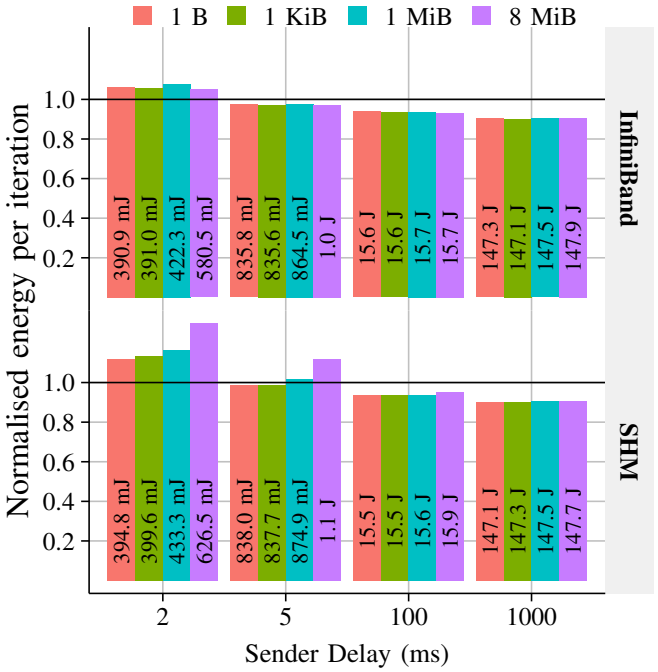
As a next step we plan to look more closely into the exact causes for the latency overhead of event-based communication in Open MPI. We surmise that parts of the overhead stem from UCX not being optimised for fast event-based communication. We also intent to better integrate event-based communication into Open MPI to avoid the explicit timeout and combine our setup with oversubscription approaches using our current findings for tuning.

Acknowledgments

We thank our anonymous reviewers for their helpful feedback and the Centre for Information Services and High Performance Computing (ZIH) TU Dresden for providing its facilities. This research was co-financed by the Federal Ministry of Education and Research of Germany in the programme of "Souverän. Digital. Vernetzt." (joint project 6G-life, project ID 16KISK001K), from the European Union's Horizon 2020 research programme under grant agreement No. 957216 and by public funding from the state of Saxony/Germany.



(a) Energy Consumed by CPU Running Rank 1



(b) Energy Consumed by Entire Node

Figure 8: Energy Consumption. The bars show the energy consumed by event-based communication normalised to the energy consumed by polling. The numbers inside the bars are the absolute energy values for event-based communication. All values refer to a single ping-pong operation in the Libra benchmark to allow for the comparison of runs with different iteration counts.

References

[1] Message Passing Interface Forum, "MPI: A message-passing interface standard," University of Tennessee, Knoxville, TN, USA, Standard 3.1, Jun. 2015. [Online]. Available: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf (visited on 01/21/2020).

[2] *Sched_yield(2) — Linux manual page*. [Online]. Available: https://man7.org/linux/man-pages/man2/sched_yield.2.html (visited on 08/01/2022).

[3] J. Schuchart, D. Hackenberg, R. Schöne, T. Ilsche, R. Nagappan, and M. K. Patterson, "The shift from processor power consumption to performance variations: Fundamental implications at scale," *Computer science-research and development*, vol. 31, no. 4, pp. 197–205, 2016. DOI: 10.1007/s00450-016-0327-2.

[4] L. G. Valiant, "A bridging model for parallel computation," *Communications of the acm*, vol. 33, no. 8, pp. 103–111, Aug. 1990, ISSN: 0001-0782. DOI: 10.1145/79173.79181.

[5] V. Marjanović, J. Labarta, E. Ayguadé, and M. Valero, "Overlapping communication and computation by using a hybrid MPI/SMPSs approach," in *Proceedings of the 24th ACM international conference on supercomputing*, ser. ICS '10, Tsukuba, Ibaraki, Japan: ACM, Jun. 2010, pp. 5–16, ISBN: 978-1-4503-0018-6. DOI: 10.1145/1810085.1810091.

[6] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer, "An energy efficiency feature survey of the Intel Haswell processor," in *Proceedings of the IEEE international parallel and distributed processing symposium workshop*, Hyderabad, India: IEEE, May 2015, pp. 896–904, ISBN: 978-1-4673-7684-6. DOI: 10.1109/IPDPSW.2015.70.

[7] H. Weisbach, B. Gerofi, B. Kocoloski, H. Härtig, and Y. Ishikawa, "Hardware performance variation: A comparative study using lightweight kernels," in *Proceedings of the 33rd international conference on high performance computing*, R. Yokota, M. Weiland, D. Keyes, and C. Trinitis, Eds., Frankfurt, Germany: Springer International Publishing, Jun. 2018, pp. 246–265, ISBN: 978-3-319-92040-5. DOI: 10.1007/978-3-319-92040-5_13.

[8] TOP500.org, "TOP500." [Online]. Available: https://www.top500.org/lists/top500/2022/06/ (visited on 09/20/2022).

[9] Douglas B. Kothe, "The Exascale Computing Project," presented at the HPC User Forum, Sep. 7, 2021. [Online]. Available: https://www.hpcuserforum.com/wp-content/uploads/2021/09/Exascale-Computing-Project-Update_ORNL_D.Kothe_Sept-2021-HPC-UF.pdf (visited on 09/23/2022).

[10] The MVAPICH team, *MVAPICH2*. [Online]. Available: https://mvapich.cse.ohio-state.edu/ (visited on 08/01/2022).

[11] E. Gabriel, G. E. Fagg, G. Bosilca, *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds., Berlin, Heidelberg: Springer, Sep. 2004, pp. 97–104, ISBN: 978-3-540-30218-6. DOI: 10.1007/978-3-540-30218-6_19.

[12] P. Shamis, M. G. Venkata, M. G. Lopez, *et al.*, "UCX: An open source framework for HPC network APIs and beyond," presented at the IEEE 23rd Annual Symposium on High-Performance Interconnects, ser. HOTI '15, Santa Clara, CA, USA: IEEE, Aug. 2015, pp. 40–43. DOI: 10.1109/HOTI.2015.13.

[13] *Unix(7) — Linux manual page*. [Online]. Available: https://www.man7.org/linux/man-pages/man7/unix.7.html (visited on 08/01/2022).

[14] *Poll(2) — Linux manual page*. [Online]. Available: https://www.man7.org/linux/man-pages/man2/poll.2.html (visited on 08/01/2022).

[15] G. Utrera, S. Tabik, J. Corbalan, and J. Labarta, "A job scheduling approach for multi-core clusters based on virtual malleability," presented at the European Conference on Parallel Processing, C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, Eds., ser. Euro-Par '12, Springer Berlin Heidelberg, 2012, pp. 191–203, ISBN: 978-3-642-32820-6. DOI: 10.1007/978-3-642-32820-6_20.

[16] G. Utrera, J. Corbalan, and J. Labarta, "Scheduling parallel jobs on multicore clusters using CPU oversubscription," *The journal of supercomputing*, vol. 68, no. 3, pp. 1113–1140, 2014.

[17] G. Utrera, M. Farreras, and J. Fornes, "Task Packing: Efficient task scheduling in unbalanced parallel programs to maximize CPU utilization," *Journal of parallel and distributed computing*, vol. 134, pp. 37–49, Dec. 2019, ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2019.08.003.

[18] Q. Xiong, E. Ates, M. C. Herbordt, and A. K. Coskun, "Tangram: Colocating HPC applications with oversubscription," in *Proceedings of the 22nd annual IEEE high performance extreme computing conference*, 2018.

[19] F. Zheng, H. Yu, C. Hantas, *et al.*, "GoldRush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution," in *Proceedings of the international conference on high performance computing, networking, storage and analysis*, ser. SC '13, [object Object], Denver, CO, USA, Nov. 2013, pp. 1–12, ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2503279.

[20] F. Wende, T. Steinke, and A. Reinefeld, "The impact of process placement and oversubscription on application performance: A case study for exascale computing," in *Proceedings of the 3rd international conference on exascale applications and software*, ser. EASC '15, Edinburgh, UK: University of Edinburgh, Apr. 2015, pp. 13–18, ISBN: 978-0-9926615-1-9.

[21] A. Frank, T. Süß, and A. Brinkmann, "Effects and benefits of node sharing strategies in HPC batch systems," in *Proceedings of the 33rd IEEE international parallel and distributed processing symposium*, ser. IPDPS '19, 2019, pp. 43–53, ISBN: 978-1-72811-246-6. DOI: 10.1109/IPDPS.2019.00016.

[22] A. Venkatesh, A. Vishnu, K. Hamidouche, *et al.*, "A case for application-oblivious energy-efficient MPI runtime," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, ser. SC '15, ACM, Austin, TX, USA: IEEE, Nov. 2015, pp. 1–12, ISBN: 978-1-4503-3723-6. DOI: 10.1145/2807591.2807658.

[23] B. Rountree, D. K. Lownenthal, B. R. De Supinski, M. Schulz, V. W. Freeh, and T. Bletsch, "Adagio: Making DVS practical for complex HPC applications," in *Proceedings of the 23rd international conference on supercomputing*, ser. ICS '09, Association for Computing Machinery, Jun. 2009, pp. 460–469, ISBN: 978-1-60558-498-0. DOI: 10.1145/1542275.1542340.

[24] A. Kulkarni and A. Lumsdaine, "A comparative study of asynchronous many-tasking runtimes: Cilk, Charm++, ParalleX and AM++," *Corr*, vol. abs/1904.00518, 2019. [Online]. Available: http://arxiv.org/abs/1904.00518.

[25] M. Lewis and A. Grimshaw, "The core Legion object model," in *Proceedings of 5th IEEE international symposium on high performance distributed computing*, ser. HPDC '96, Syracuse, NY, USA: IEEE, Aug. 1996, pp. 551–561, ISBN: 0-8186-7582-9. DOI: 10.1109/HPDC.1996.546226.

[26] P. Charles, C. Grothoff, V. Saraswat, *et al.*, "X10: An object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications*, ser. OOPSLA '05, ACM, vol. 40, San Diego, CA, USA: Association for Computing Machinery, 2005, pp. 519–538, ISBN: 1-59593-031-0. DOI: 10.1145/1094811.1094852.

[27] G. Martín, M.-C. Marinescu, D. E. Singh, and J. Carretero, "FLEX-MPI: An MPI extension for supporting dynamic load balancing on heterogeneous non-dedicated systems," in *Euro-par 2013 parallel processing*, F. Wolf, B. Mohr, and D. an Mey, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 138–149, ISBN: 978-3-642-40047-6. DOI: 10.1007/978-3-642-40047-6_16.

[28] J. Weidendorfer, D. Yang, and C. Trinitis, "LAIK: A library for fault tolerant distribution of global data for parallel applications," presented at the PARS, ser. PARS '17, Hagen, 2017, p. 10. [Online]. Available: https://mediatum.ub.tum.de/doc/1375185/1375185.pdf.

[29] D. Stark, R. F. Barrett, R. Grant, S. L. Olivier, K. Pedretti, and C. T. Vaughan, "Early experiences co-scheduling work and communication tasks for hybrid MPI+X applications.," presented at the Workshop on Exascale MPI at Supercomputing Conference, ser. ExaMPI '14, Sep. 2014, pp. 9–19. DOI: 10.1109/ExaMPI.2014.6.

[30] C. Huang, O. Lawlor, and L. V. Kalé, "Adaptive MPI," in *Lecture Notes in Computer Science*, L. Rauchwerger, Ed., ser. LCPC 2003, vol. 2958, Springer, 2003, pp. 306–322, ISBN: 978-3-540-21199-0. DOI: 10.1007/978-3-540-24644-2_20.

[31] H. Kamal and A. Wagner, "An integrated fine-grain runtime system for MPI," *Computing*, vol. 96, no. 4, pp. 293–309, 2014. DOI: 10.1007/s00607-013-0329-x.

[32] Patrick Mochel and Mike Murphy, *Sysfs - _The_ filesystem for exporting kernel objects*. [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/sysfs.html (visited on 08/01/2022).

[33] The Ohio State University's Network-Based Computing Laboratory. "OSU micro-benchmarks," MVAPICH :: Benchmarks. (2001–2018), [Online]. Available: http://mvapich.cse.ohio-state.edu/benchmarks/ (visited on 08/02/2022).

[34] D. Hackenberg, T. Ilsche, J. Schuchart, *et al.*, "HDEEM: High definition energy efficiency monitoring," in *Proceedings of the 2nd international workshop on energy efficient supercomputing*, ser. E2SC '14, New Orleans, LA, USA: IEEE Press, Nov. 2014, pp. 1–10, ISBN: 978-1-4799-7036-0. DOI: 10.1109/E2SC.2014.13.