SAND2020-3303C

The Case for Explicit Reuse Semantics for RDMA Communication

Scott Levy
Sandia National Laboratories
Center for Computing Research
Albuquerque, NM, USA
sllevy@sandia.gov

Patrick Widener
Sandia National Laboratories
Center for Computing Research
Albuquerque, NM, USA
patrick.widener@sandia.gov

Craig Ulmer
Sandia National Laboratories
Livermore, CA, USA
cdulmer@sandia.gov

Todd Kordenbrock
Perspecta, Inc.
thkorde@sandia.gov

***Abstract*—Remote Direct Memory Access (RDMA) is an increasingly important technology in high-performance computing (HPC). RDMA provides low-latency, high-bandwidth data transfer between compute nodes. Additionally, it does not require explicit synchronization with the destination processor. Eliminating unnecessary synchronization can significantly improve the communication performance of large-scale scientific codes. A long-standing challenge presented by RDMA communication is mitigating the cost of registering memory with the network interface controller (NIC). Reusing memory once it is registered has been shown to significantly reduce the cost of RDMA communication. However, existing approaches for reusing memory rely on implicit memory semantics. In this paper, we introduce an approach that makes memory reuse semantics explicit by exposing a separate allocator for registered memory. The data and analysis in this paper yield the following contributions: (i) managing registered memory explicitly enables efficient reuse of registered memory; (ii) registering large memory regions to amortize the registration cost over multiple user requests can significantly reduce cost of acquiring new registered memory; and (iii) reducing the cost of acquiring registered memory can significantly improve the performance of RDMA communication. Reusing registered memory is key to high-performance RDMA communication. By making reuse semantics explicit, our approach has the potential to improve RDMA performance by making it significantly easier for programmers to efficiently reuse registered memory.**

***Index Terms*—RDMA, HPC, memory management, messaging**

I. Introduction

Modern extreme-scale computational science is supported by high-performance computing (HPC) systems designed around low-latency, high-bandwidth networks. Remote Direct Memory Access (RDMA) leverages the capabilities of these high-performance networks by providing zero-copy messaging between a specialized programmable NIC and an application without direct involvement from the target processor. In order to realize these benefits on modern interconnects, an application must *register* a memory region on both sides of a message exchange with the NIC. However, memory registration is an expensive operation. As a result, programmers have focused on reusing memory buffers once they have been registered. A common approach is to cache registered memory regions so that subsequent RDMA operations can reuse registered memory. However, these cache-based approaches typically rely on *implicit* semantics: the application programmer has no explicit knowledge of whether a given buffer is still registered (i.e., whether it is still resident in the cache). Instead the programmer hopes that by carefully constructing her program, her memory buffers will still be registered when she initiates the next RDMA operation.

In this paper, we propose an approach that presents the programmer with *explicit* reuse semantics for memory buffers. We make and empirically support the following specific claims: (i) explicit management of registered memory facilitates its efficient reuse; (ii) registration of large memory regions amortizes the registration cost over multiple user requests and can significantly reduce the cost of registering new memory; and (iii) reducing the cost of acquiring registered memory can improve the performance of RDMA communication.

Our discussion is organized as follows. We motivate our proposed approach in Section II. We describe our approach in Section III and provide background on related software in Section IV. Sections V and VI provide detail on our experimental environment, describe the microbenchmarks we use to support our performance claims, and discuss our experimental results. Section VII examines the performance of our approach in the context of a case study of using RDMA communication with the Kokkos programming framework. Sections VIII and IX discuss related work and conclude the paper, respectively.

II. The Importance of Reusing Registered Memory

The cost of registering memory for RDMA communication has long been recognized as a significant performance issue. As a result, mitigation methods have been well-studied. The most common approach is to build an application-specific cache to facilitate reuse of registered memory. These approaches can either be dynamic (i.e., caching buffers as requests arrive) [1], [2], [3], [4], [5] or static (i.e., pre-allocating buffers during initialization) [6], [7]. However, many of these approaches are built within specific applications and rely on implicit semantics of memory buffer reuse or predictions about future memory buffer use. Implicit reuse semantics

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

mean that application programmers cannot guarantee that any given memory buffer remains registered (i.e., they cannot guarantee that any given registered memory buffer is still in the registration cache). Instead, in order to achieve high performance, the programmer is compelled to carefully manage the reuse of her memory buffers in the hope that they remain in a registration cache. In this paper, we propose to make buffer reuse semantics explicit to ease the burden on application programmers. The basic idea, explained in more detail in Section III, is to allow the programmer to explicitly acquire and release regions of registered memory for RDMA communication.

III. Overview of Proposed Approach

On many modern HPC interconnects (e.g., Cray Aries, Infiniband), RDMA requires registration of the source and destination memory regions. Registration requires requesting the kernel to pin the memory (i.e., ensure that the region is in physical memory and cannot be swapped) and then communicating information about this pinned memory region to the NIC. The time required to register memory for RDMA communication has long been recognized as a significant overhead in high-performance communication. Standard practice is to reduce this cost by reusing memory that has already been registered. However, managing registered memory to ensure reuse is tedious and error-prone. The motivation for managing registered memory to facilitate reuse is substantially similar to the argument for using standard memory allocators (e.g., `malloc`, `memalign`) for managing the reuse of heap memory acquired from the kernel (e.g., using `sbrk` or `brk`) rather than requiring to programmers to carefully construct their programs to ensure efficient reuse.

In this paper, we introduce an approach to address these issues. Our proposed approach uses two semantically distinct allocators: one for standard heap memory; and one for registered memory. As a result, the programmer explicitly knows whether the memory for a given buffer is registered or not. Moreover, by managing registered memory explicitly we can provide strong guarantees to the programmer that it is efficiently reused. This approach provides two principal benefits: (i) it frees programmers from managing RDMA buffer reuse explicitly by making memory semantics explicit; and (ii) it reduces the cost of acquiring new registered memory by allocating large blocks of memory to amortize the registration cost over multiple user requests. It is still possible to register regions of memory acquired from the standard heap memory allocators, but by allowing the programmer to be explicit about the type of memory that they need, this approach can provide lower-latency RDMA operations.

Fundamentally, our approach presents the programmer with two semantically distinct memory allocators: the standard memory allocators (e.g., `malloc`, `memalign`) that allocate ordinary memory; and a new memory allocator (our modified version of TCMalloc) that manages memory that has been pre-registered to improve RDMA performance. It is still possible to register regions of memory acquired from the standard memory allocators, but by allowing the programmer to be explicit about the type of memory that they need, our approach can facilitate lower-latency RDMA operations.

In this initial implementation, we manage registered memory with a customized version of TCMalloc.[1] However, our approach does not depend on any particular feature of TCMalloc. In principle, any third-party memory allocator (e.g., Hoard [8], jemalloc [9], [9]) could be used for this purpose.

To prevent conflicts between TCMalloc and the default memory allocators, we have modified TCMalloc so that it uses `memalign` to acquire new memory. The minimum number of bytes of memory that TCMalloc will request is 1 MiB. Additionally, TCMalloc immediately registers each new block of memory that it acquires.

IV. Background

A. NNTI

The Nessie Network Transport Interface (NNTI) provides a portable, lightweight abstraction for RDMA operations across HPC interconnects [10], [11]. For our purposes, NNTI provides a single interface for RDMA communication over multiple interconnects. It includes support for Infiniband and Cray XC Series interconnects; Infiniband Verbs (`libibverbs`) is used for Infiniband networks and Cray's User-Level Generic Network Interface (uGNI) is used for Cray interconnects.

NNTI is part of Faodel [12], an open-source software package developed at Sandia National Laboratories that provides mechanisms for managing data movement between related jobs running on HPC platforms. The Faodel source code, including NNTI, is publicly available on GitHub [13].

B. Kokkos

The proliferation of multi-core processors has presented both new opportunities and new complexities for application developers. The Kokkos [14] programming model provides abstractions to insulate software developers from hardware details, while providing performance portability across many architectures. Kokkos is a C++ library that is linked with the target application.

The key Kokkos abstractions that we rely on for the case study presented in this paper are: *memory spaces* and `Views`. A Kokkos memory space is essentially a memory domain (e.g., host memory, GPU memory) and an associated allocator. In order to investigate the potential performance impact of using our approach to facilitate RDMA transfers of Kokkos data structures between nodes, we created a new memory space, `PinnedMemorySpace` for use in our tests.

A `View` encapsulates a multidimensional array in the context of a memory space. Our `PinnedMemorySpace` enables programmers to create `Views` directly from registered memory. More details and the results of our `View` experiments are presented in Section VII.

[1] TCMalloc was designed as a drop-in replacement for `malloc`, i.e., it assumes that it is solely responsible for managing heap memory. We have modified it to remove these assumptions.

	Stria	Mutrino
Architecture	Linux Infiniband cluster	Cray XC40
Interconnect	Mellanox IB (ConnectX5)	Cray Aries
Processor	Marvell ThunderX2	Intel Xeon (Haswell)

TABLE I
COMPUTING SYSTEMS USED FOR EXPERIMENTS

V. EXPERIMENTAL APPROACH

In this paper, we use microbenchmarks and a case study to explore RDMA communication on HPC platforms. Our data was collected on two computing platforms at Sandia National Laboratories: *Mutrino* and *Stria*. Mutrino is a Cray development system for Trinity [15]. Although it is much smaller than Trinity, its system administrators strive to ensure that its hardware and software configuration matches Trinity. Like Trinity, Mutrino has two compute node partitions: one consisting of nodes built around Intel Haswell processors and one consisting of nodes built around Intel Knights Landing processors. The Mutrino results in this paper are from the Haswell partition. Mutrino's interconnect is Cray's Aries network. Similiarly, Stria is a development system for Astra. Astra is a high-performance computing cluster built on ARM-based processors.[2] Stria has an Infiniband network that uses Mellanox ConnectX5 Infiniband NICs. Both systems use 4 KiB memory pages.[3] Details of these systems are summarized in TABLE I.

All of our microbenchmark results (*see* Section VI) were obtained using NNTI to provide RDMA communication. For the experiments performed on Mutrino, we configured NNTI to use uGNI with Mutrino's Aries network. For the experiments performed on Stria, we configured NNTI to use Infiniband Verbs (`libibverbs`).

VI. MICROBENCHMARK RESULTS

In this section, we use a set of microbenchmarks to explore the potential benefits of using our proposed approach to facilitate reuse of registered memory.

A. Experimental Design

For each of the microbenchmark experiments in this section, we performed $1,000$ trials for each of 15 memory buffer sizes: 2^k bytes, $7 \leq k \leq 21$ (i.e., powers-of-two ranging from 128 bytes to 2 MiB). The time required to complete each trial was independently measured. Each set of trials was performed on a single allocation obtained from the resource manager.

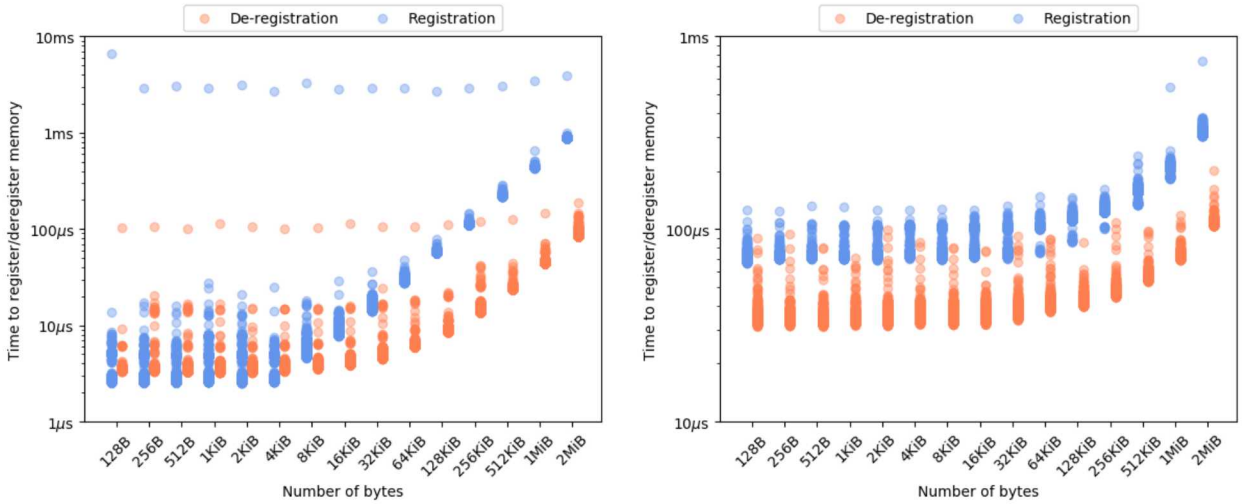
[2]Astra debuted at number 203 on the Top500 list in November 2018 [16] and was the first ARM-based supercomputer to qualify for the Top500 [17].

[3]Throughout this paper, we use the binary prefixes defined by the International Electrotechnical Commission (IEC) to indicate binary orders of magnitude. For example, a KiB is a *kibibyte*, 2^{10} bytes and MiB is a *mebibyte*, 2^{20} bytes.

B. Cost of Registering and De-registering Memory

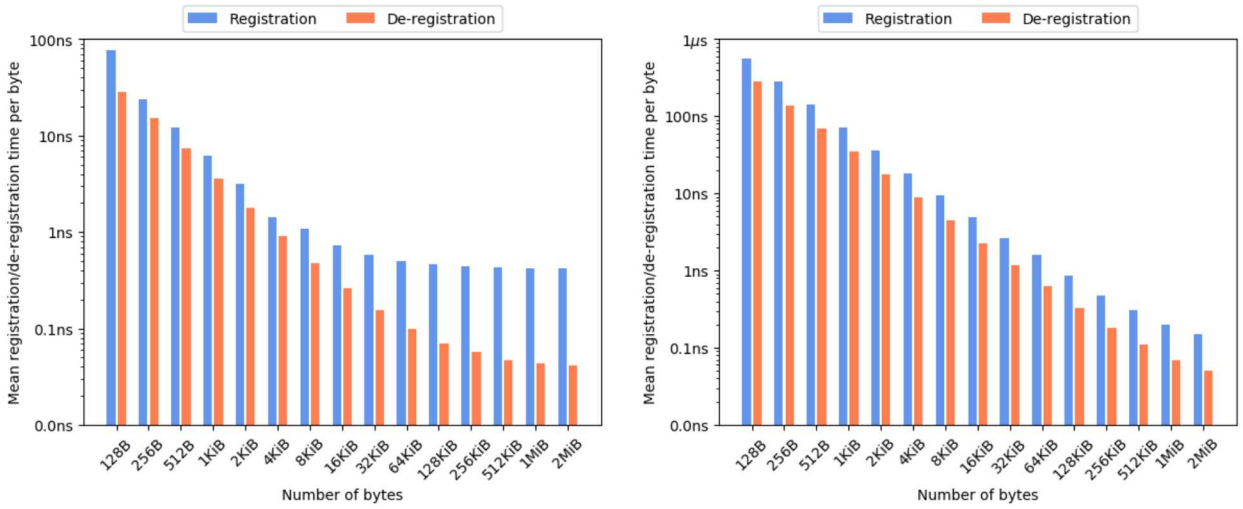
RDMA on modern interconnects requires registration of the source and destination memory regions, *see* Section IV. In this subsection, we examine the time required to register and de-register memory on our two systems. To measure registration and de-registration time, we constructed a simple microbenchmark. For each trial, it allocates, registers and de-registers a memory buffer. We recorded the time to register and de-register the memory for each trial. More details on the trials conducted are provided in Section VI-A. The results of these experiments are shown in Fig. 1. Each circle represents the results from a single trial: pale blue for registration results and pale orange for de-registration results. The position of each circle on the x-axis corresponds to the size of the allocated memory buffer. The position of each circle on the y-axis corresponds to the amount of time required to acquire the memory buffer. Darker regions form when the results of multiple experiments overlap. These results demonstrate that registration generally requires more time than de-registration and that the time required to register or de-register memory increases with the number of bytes being registered. Registering small buffers is generally faster on Mutrino, but registering large buffers is generally faster on Stria. Additionally, for both systems the registration/de-registration times can vary significantly across trials. In particular, Fig. 1a shows that for a small number of trials on Mutrino, memory registration required several milliseconds to complete, independent of the number of bytes that were registered. These outliers correspond to the result from the first trial for each memory block size. In addition, the very first registration attempt (which registered a 128-byte buffer), required more time to complete than any other trial. As a result, we believe that these results are inflated due to initialization costs that are not incurred by subsequent trials. Although there is variation in the registration costs for Stria, which uses an ARM processor and an Infiniband interconnect, the data collected from it do not exhibit the same outlier behavior.

On both systems, the time required to register or de-register memory grows significantly with the size of the target memory. On Mutrino, the mean time required to register a 2 MiB memory buffer was approximately $90\times$ slower than registering a 128-byte memory buffer. On Stria, registering a 2 MiB memory buffer was approximately $4.4\times$ slower than the mean time required to register 128 bytes. However, because the time required to register a memory buffer grows more slowly than the size of the buffer, it is much more efficient to register one large memory region than several smaller regions. Fig. 2 shows the mean registration and de-registration times per byte as a function of the memory buffer size. These data show that on Stria the per-byte registration and de-registration times decreases nearly linearly as a function of the size of the target memory. On Mutrino, the registration cost-per-byte plateaus for large ($\geq$ 128KiB) memory buffers.



(a) Mutrino (b) Stria

Fig. 1. **Time required to register memory.** Each semi-transparent circle represents the results of a single trial, darker regions occur where multiple results overlap. Note that the y-axis is discontinuous in both figures.



(a) Mutrino (b) Stria

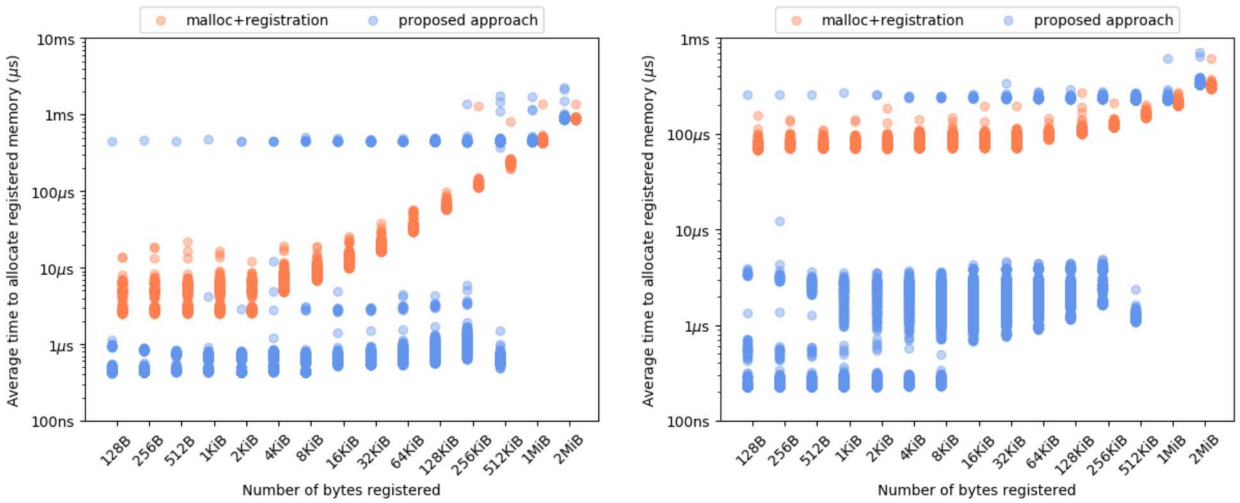
Fig. 2. **Per-byte registration and de-registration cost.** These data show the mean per-byte registration and de-registration time as a function of the size of the memory buffer that is being registered.

C. Cost of Allocating New Registered Memory

In this subsection, we examine the time required to allocate new blocks of registered memory using our proposed approach and compare it to the time required to acquire memory from a standard memory allocator (i.e., `malloc`) and explicitly register it. Allocating *new* registered memory means that satisfying a user request requires increasing the total number of bytes that are currently registered.[4] To compare these two approaches we wrote a simple microbenchmark that records the time required to acquire a region of registered memory using our approach and using allocation plus explicit registration. Because the objective of the experiments that use this benchmark is measure how long it takes to acquire registered memory without reuse, none of the allocated memory is released until all of the trials complete. More details on the trials conducted are provided in Section VI-A. The results of these experiments are shown in Figures 4 and 3.

In Fig. 3, the result of each experiment is represented by a single pale blue circle (for results that use our proposed approach to explicitly manage registered memory) or a single pale orange circle (for `malloc` and explicit registration). The position of each circle on the x-axis corresponds to the size

[4]When registered memory is being reused (*cf.* Section VI-D), an existing region of registered memory is being used, so the number of bytes of registered memory do not change.



(a) Mutrino

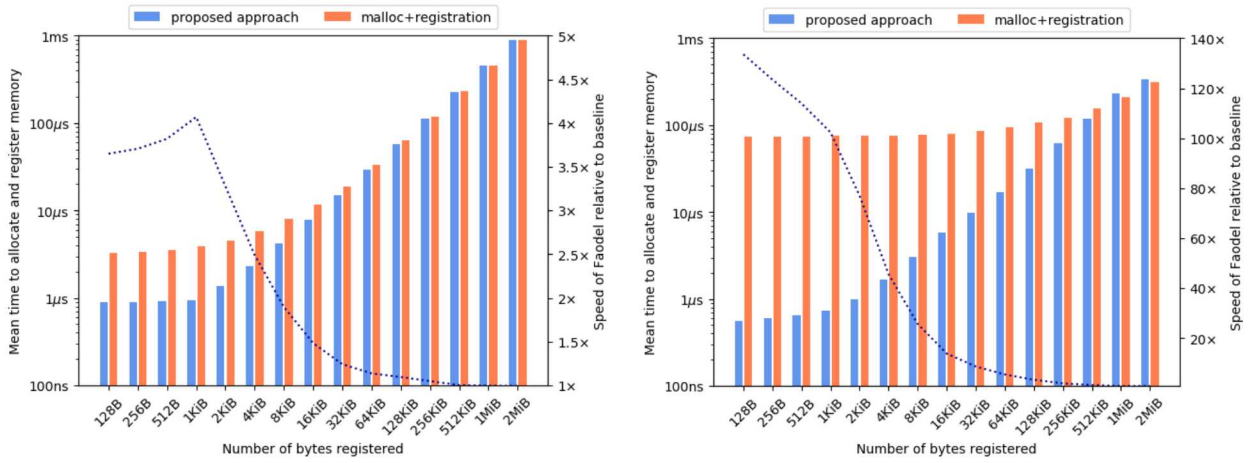
(b) Stria

Fig. 3. **Time required to allocate registered memory.** Each semi-transparent circle represents the results of a single trial, darker regions occur where multiple results overlap.

of the allocated memory buffer. The position of each circle on the y-axis corresponds to the amount of time required to acquire the memory buffer. Darker regions form when the results of multiple experiments overlap. On the whole, the results for the explicit registration experiments form a curve that is very similar to the registration results in Fig. 1 (once the differences in the scale of the figures are accounted for), thereby confirming that the cost of allocating registered memory are dominated by the time required to register the memory. The results from the experiments using our proposed approach exhibit a different phenomenon. A significant majority of these experiments complete much faster than the explicit registration experiments. Additionally, the speed of the fast experiments ($< 10\mu$s) remains relatively constant as the size of the memory region increases. However, several of the trials with our proposed approach require hundreds of microseconds to complete. The frequency of these slow results increases as the size of the memory buffer that is being allocated increases. For memory regions that are 1 MiB or greater in size, there are no fast trials. All of the trials for these regions take hundreds of microseconds to complete, much like the corresponding explicit registration experiments. The results of the experiments with our proposed approach are explained by examining how TCMalloc handles allocation requests. When TCMalloc receives an allocation request that cannot be satisified using registered memory that it already manages, it allocates new memory (using `memalign`) to satisfy the request and then registers it using NNTI. The minimum number of bytes that TCMalloc will request is 1 MiB. For small allocations (128 to 512 bytes), a single 1 MiB allocation is sufficient to provide the necessary memory for all of the trials for a specific memory buffer size. For memory regions between 1 KiB and 256 KiB, TCMalloc must request multiple allocations to satisfy all of the user allocation requests. For memory regions that are 1 MiB or greater in size, every user allocation request requires TCMalloc to acquire and register more memory. Each such acquisition obtains exactly enough 8 KiB memory blocks[5] to satisfy the user's request. The slow trials correspond to cases where TCMalloc is requesting and registering new memory. When the size of the allocated memory region is equal to or larger than the size of TCMalloc's minimum system allocation request, its performance devolves to the explicit registration case since it requests exactly the amount of memory that is necessary to satisfy the current user request (i.e., for memory buffers that are 1 MiB or larger and cannot be allocated from existing registered memory), TCMalloc's approach is effectively the same as explicit registration).

Fig. 4 shows the mean time required to allocate a block of registered memory as a function of its size. The blue bars represent the results of the trials that use our proposed approach and the orange bars represent the results of the trials that allocate and explicitly register memory. The height of each bar represents the mean allocation time as shown on the left y-axis. The dotted line is plotted relative to the right y-axis and shows how fast allocation with our approach is relative to allocation and explicit registration. For memory regions that are much smaller than TCMalloc's minimum system allocation request (e.g., 128 KiB or smaller), our approach is dramatically faster than explicit registration: more than $134\times$ faster for 128-byte memory buffers on Stria and more than $3.4\times$ faster for 128-byte memory buffers on Mutrino. The speed of our approach is due to its exploitation of the fact that the per-byte registration costs decrease as the size of the allocated memory region increases, *cf.* Section VI-B. However, as the size of the memory buffer approaches the size of TCMalloc's minimum

[5]Internally, TCMalloc manages memory in 8 KiB blocks



(a) Mutrino

(b) Stria

Fig. 4. **Mean time required to allocate new registered memory.** Each bar represents the arithmetic mean of the time required to complete an allocation that requires new memory registered (i.e., when it is not possible to reuse an existing registered memory buffer). The dashed line represents how much faster memory allocation is with our proposed approach relative to standard memory allocation and explicit registration.

system allocation request (i.e., 1 MiB), the benefits diminish. Above this threshold, the advantages of using our approach to obtain new blocks of registered memory vanish[6] because its approach essentially degrades to explicit registration.[7]

In addition to making reuse semantics explicit, our approach has the potential to significantly reduce the amount of time required to obtain new blocks of registered memory. If TCMalloc can satisfy a user's request from memory it already manages, it is exceptionally fast: generally less than 10μs, *cf.* Fig. 3. Similarly, if the size of a new memory buffer is much smaller than the size of TCMalloc's minimum system allocation request, our approach is able to outperform explicit registration by amortizing the cost of memory registration across multiple user allocation requests.

D. Cost of Reusing Registered Memory

As we have stated several times in this paper, the principal benefit of our approach is to facilitate and simplify the reuse of registered memory. In this subsection, we compare the cost of reusing registered memory obtained from TCMalloc with the cost of acquiring ordinary heap memory and explicitly registering it. To measure the cost of satisfying a user request with memory that has already been registered, we constructed a benchmark that includes a warm-up loop to ensure that TCMalloc is managing enough registered memory to satisfy all subsequent requests in the experiment. For each trial, we measure the amount of time required to acquire a memory buffer using TCMalloc to reuse existing registered memory. We also measured the time required to acquire a memory buffer using `malloc` and explicit registration. Comparing the results of these two sets of experiments allows us to characterize the benefit of using our proposed approach to reuse memory. More details on the trials that we performed are provided in Section VI-A. The results of these experiments are shown required to obtain a block of registered memory as a function of its size is shown in Fig. 5.

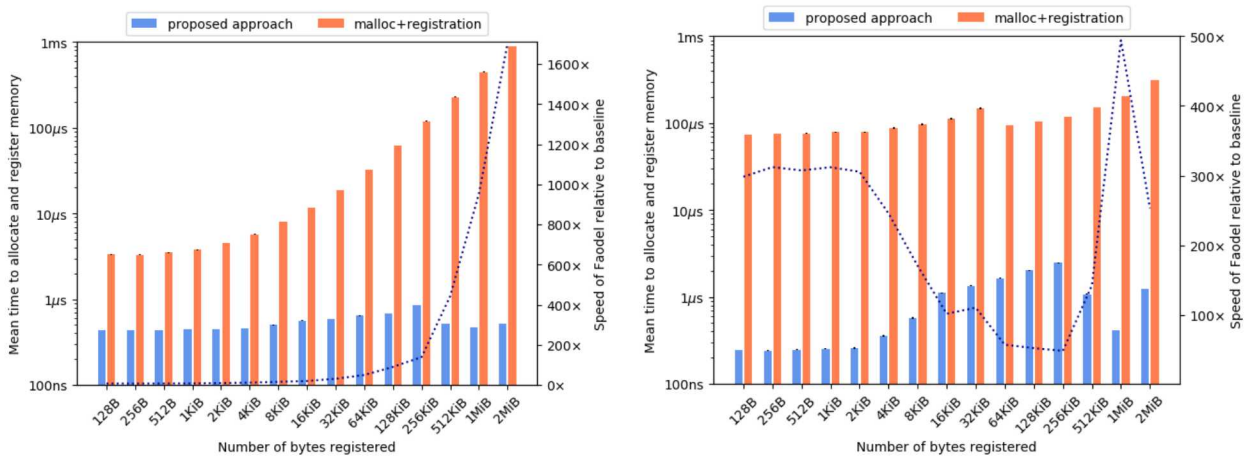
The data presented in Fig. 5 demonstrate the relative speed of reusing registered memory and obtaining and explicitly registering new memory. Each bar is plotted relative to the left y-axis and its height corresponds to the mean time required to acquire a memory buffer. The blue bars represent the time to acquire a memory buffer using our proposed approach and the orange bars represent the time to allocate and explicitly register the memory buffer. The dotted line is plotted relative to the right y-axis and shows the performance of our proposed approach relative to allocation and explicit registration.

On Mutrino, reusing memory was very fast: the mean time to acquire a memory buffer was less than 1μs. In contrast, the mean time required to explicitly register a new memory buffer was as much as 883μs. Reusing memory can therefore reduce the mean time to acquire a registered memory buffer by as much as 882μs. In relative terms, reusing memory was up to $2054\times$ faster than explicit registration of a new buffer.

Similarly, on Stria, reusing registered memory was very fast: the mean time to acquire a memory buffer was less than 2.5μs. In contrast, the mean time to allocate and explicitly register a memory buffer was as much as 311μs. Reusing memory can therefore reduce the mean time to acquire a registered memory buffer by more than 310μs. In relative terms, reusing registered memory is up to $253\times$ faster than allocation and explicit registration.

[6]In fact, the data collected on Stria for the very largest memory buffers shows that there is a modest penalty (up to 11.3%) for using TCMalloc to allocate new registered memory for a buffer that is 1 MiB or larger in size. *However*, because our approach is intended to facilitate efficient reuse of registered memory, this modest one-time penalty for large memory buffers will not significantly diminish the performance benefit of reusing registered memory, *cf.* Fig. 5 (showing that reusing a large buffer is nearly 900μs faster than allocating and explicitly registering a new memory buffer)

[7]The principle advantage of our approach, facilitating reuse of registered memory, is unaffected by the number of bytes obtained.



(a) Mutrino

(b) Stria

Fig. 5. **Mean time required to reuse registered memory.** The height of each bar represents the arithmetic mean of the time required to obtain a region of registered memory.

E. RDMA Put

In this subsection, we examine the impact of our proposed approach on the time required to transfer data between nodes using RDMA Put operations. To measure the benefit of reusing registered memory with our approach, we constructed a benchmark that includes a warm-up loop to ensure that TCMalloc is managing enough registered memory to satisfy all subsequent requests in the experiment. The benchmark uses either our proposed approach to acquired a memory buffer or allocates heap memory for the buffer and explicit registers it. It then transfers the buffer's contents from one node to another using an RDMA Put operation. More details on the trials that we performed are provided in Section VI-A. The results of this benchmark allow us to examine the benefit of using our proposed approach relative to allocating and explicitly registering memory buffers.

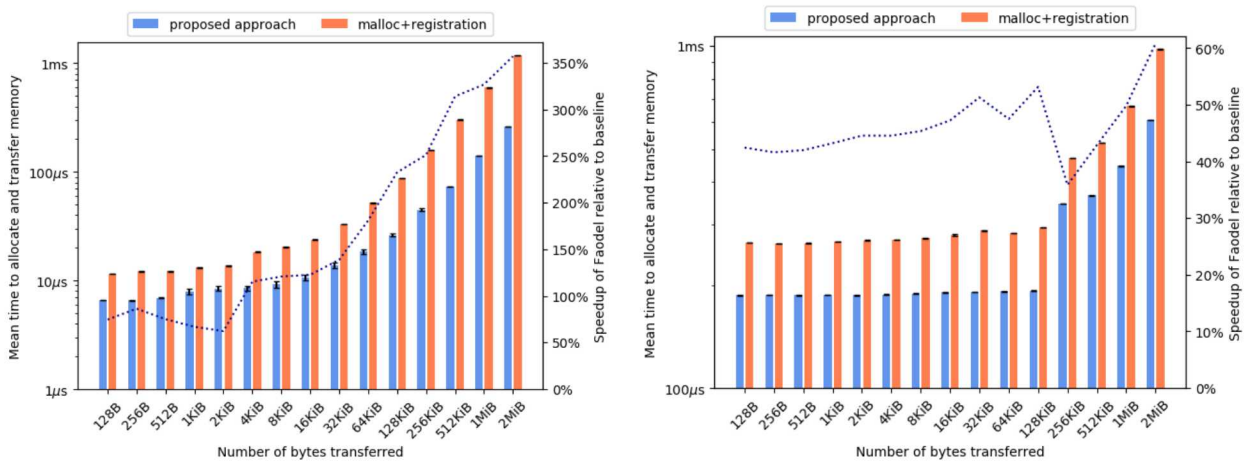
The results of these experiments are shown in Fig. 6. The bars are plotted relative to the left y-axis and their height represents the mean time to acquire and transfer the contents of a memory buffer using an RDMA Put operation. The blue bars represent the results of the experiments that use our proposed approach to reuse registered memory for the memory buffers. The orange bars represent the results of the experiments that allocate heap memory with `malloc` and explicitly register it. The error bars extend above and below the mean by the standard error of the mean; the error bars are scarcely visible because the standard error data is very small. The dotted line is plotted relative to the right y-axis and represents how fast using our approach is relative to the baseline. These results demonstrate that using our proposed approach to reuse registered memory is significantly faster with our approach: up to $4.5\times$ faster on Mutrino and 60.0% faster on Stria.

VII. Case Study: `Kokkos::View` + RDMA

RDMA has been well-understood for more than a decade, *cf.* [18]. Despite its long history, one-sided communication remains rare in important scientific applications; communication continues to be dominated by two-sided operations. As a result, in this section we develop an understanding the practical benefit of explicitly managing registered memory by examining results from a case study: using RDMA to transfer the contents of Kokkos `Views` between nodes.

Programming frameworks are extremely valuable to the development of scientific simulation codes. They allow programmers to focus on their problem domain and to outsource the solution of common programming issues (e.g., effective multithreading) to external libraries. Kokkos [14] is an open-source programming framework that facilitates performance portability. Programmers can write their code once and achieve high performance across different architectures (e.g., on both GPUs and CPUs). As a result, Kokkos is projected to be a key programming model for exascale, *see* [19]. However, Kokkos, like many such libraries, conceals memory allocation behind API calls. Therefore, it is not currently straightforward to construct `Views` from registered memory. In this section, we demonstrate how our approach can be integrated with Kokkos to create `Views` directly from registered memory and to efficiently transfer their contents between nodes using RDMA.

One of the key abstractions in Kokkos is the idea of a Memory Space. Different Memory Spaces represent different memory domains and encapsulate memory allocation functions. For example, `HostSpace` represents standard CPU-accessible memory and `CudaSpace` repesents memory on a GPU. To allow us to build a `View` from existing registered memory acquired, we created a new memory space: `PinnedMemorySpace`. Using this Memory Space, we constructed a microbenchmark that allows us to compare three strategies for registering the memory that stores the contents of a `View`: (i) `PinnedMemorySpace` builds a `View` on top of registered memory acquired using our apporoach;



(a) Mutrino

(b) Stria

Fig. 6. **Mean time required to transfer memory using registered memory.** The height of each bar represents the arithmetic mean of the time required to obtain a region of registered memory and transfer its contents from one node to another.

(ii) `HostSpace` builds a `View` from ordinary memory and explicitly registers its contents using a pointer to the memory obtained via an API call; and (iii) *Serialization* builds a `View` from ordinary memory and copies its contents into registered memory acquired using our approach (i.e., explicitly requesting registered memory from TCMalloc). When the contents of the `View` are in registered memory, our microbenchmark initiates an RDMA Put operation to transfer the contents of the `View` to a remote node. More details on the trials that we performed are provided in Section VI-A. The results of these experiments are shown in Fig. 7.

In Fig. 7 the height of each bar corresponds to the mean time required to transfer the contents of a `View` from one node to another using an RDMA Put operation. The error bars extend above and below the mean by the standard error of the mean; the error bars are scarcely visible because the standard error for these data is very small.

For transfers of small `Views` (128 bytes to 2 KiB), the `PinnedMemorySpace` approach is significantly faster than using `HostSpace`, as much as 77% faster on Mutrino and 42% faster on Stria. As shown in the results presented in Section VI-C, this is due to the registration cost amortization benefits that our approach provides. However, for these small buffers, using a `PinnedMemorySpace` provides a modest improvement over serialization. This is due to the fact that both approaches are leveraging our approach to explictly obtain registered memory and the cost of copying a small number of bytes into registered memory is small. For `Views` that are larger than 2 KiB using a `PinnedMemorySpace` is generally much faster than the other two approaches. On Mutrino, it was more than $9.4\times$ as fast as serialization and as much $4.2\times$ faster than `HostSpace`. On Stria, it was more than $19.4\times$ as fast as serialization and as much as 50% faster than `HostSpace`.

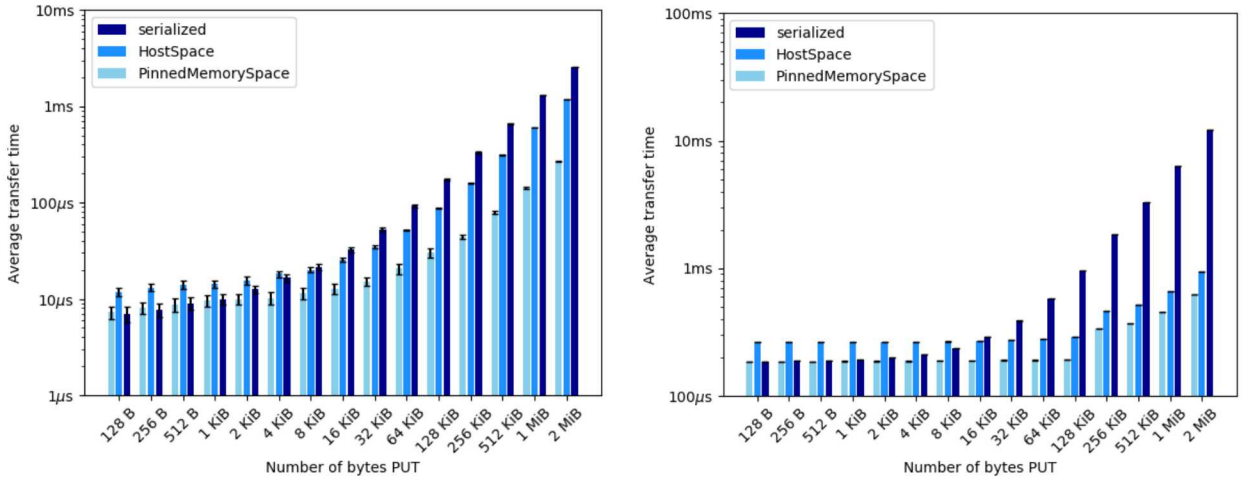
VIII. Related Work

The cost of registering memory for RDMA communication has long been recognized as a significant issue. As a result, mitigation methods have been well-studied. The most common approach is to cache registered memory [1], [2], [3], [4], [6], [7], [5].[8] Another approach is to overlap registration with other operations to help hide memory registration costs [20], [21]. The viability and benefit of overlapping memory registration with other operations is highly application dependent.

In contrast to these approaches, our approach allows programmers to explicitly express their intent to use regions of registered memory for RDMA communication (e.g., by requesting registered memory from our specialized version of TCMalloc or by requesting ordinary heap memory from `malloc`). The most similar approach can be found in the memory pools used by Jia et al. [22] for TensorFlow. These memory pools are integrated into the TensorFlow framework but the authors provide no analysis of the benefit of this approach to reusing registered memory outside of this framework. Our approach makes reuse semantics explicit by providing an application-independent service. We have implemented our approach as a stand-alone component of *Faodel* [12]. Faodel is an open-source data management software package that is publicly available, *see* [13]. Moreover, we provide a detailed analysis in this paper of the potential benefits of reusing registered memory.

Mellanox has recently introduced On-Demand Paging (ODP) in some of its Host Channel Adapters (HCA). Given the relative novelty of these devices, relatively little research on the impact of ODP on Infiniband communication performance has been published. Li et al. [23] have that there are challenges associated with effectively exploiting ODP in MPI implementations. Moreover, their proposed approach still

[8] *See* Section II for a more detailed discussion of these approaches.



(a) Mutrino (b) Stria

Fig. 7. **Comparison of time required to use RDMA to transfer the contents of a `View` from one node to another**

relies on a cache of pinned memory. As a result, there may still be an opportunity to use our proposed approach to manage pinned memory explicitly.

IX. Conclusion

RDMA-based communcation has been a preferred solution for many applications running on reliable, low-latency networks. Recent innovations (*e.g.*, [24]) from the enterprise/cloud computing space such as RDMA on converged Ethernet [25] are making RDMA solutions more attractive in other contexts as well. In this paper, we have demonstrated how explicit management of registered memory can provide clear performance gains for RDMA communication. Our approach provides a flexible and high-performance interface for acquiring and efficiently reusing registered memory.

References

[1] A. Denis, "A high performance superpipeline protocol for InfiniBand," in *European Conference on Parallel Processing*. Springer, 2011, pp. 276–287.

[2] M. J. Rashti and A. Afsahi, "Exploiting application buffer reuse to improve MPI small message transfer protocols over RDMA-enabled networks," *Cluster Computing*, vol. 14, no. 4, pp. 345–356, 2011.

[3] L. Ou, X. He, and J. Han, "An efficient design for fast memory registration in RDMA," *Journal of Network and Computer Applications*, vol. 32, no. 3, pp. 642–651, 2009.

[4] J. Wu, P. Wyckoff, and D. Panda, "PVFS over infiniband: Design and performance evaluation," in *2003 International Conference on Parallel Processing, 2003. Proceedings.* IEEE, 2003, pp. 125–132.

[5] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa, "Pin-down cache: A virtual memory management technique for zero-copy communication," in *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*. IEEE, 1998, pp. 308–314.

[6] J. Wu, P. Wyckoff, D. Panda, and R. Ross, "Unifier: unifying cache management and communication buffer management for PVFS over InfiniBand," in *IEEE International Symposium on Cluster Computing and the Grid, 2004. CCGrid 2004.* IEEE, 2004, pp. 523–530.

[7] H.-C. Lin, H.-C. Chen, and S.-C. Lin, "A pre-registered buffering method for iSER-based storage over IP SAN," in *INC2010: 6th International Conference on Networked Computing*. IEEE, 2010, pp. 1–4.

[8] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," in *ACM SIGARCH Computer Architecture News*, vol. 28, no. 5. ACM, 2000, pp. 117–128.

[9] jemalloc, "jemalloc memory allocator," https://github.com/jemalloc, accessed: 10-Jan-2020.

[10] J. Lofstead, R. Oldfield, T. Kordenbrock, and C. Reiss, "Extending scalability of collective io through nessie and staging," in *Proceedings of the sixth workshop on Parallel Data Storage*. ACM, 2011, pp. 7–12.

[11] R. A. Oldfield, P. Widener, A. B. Maccabe, L. Ward, and T. Kordenbrock, "Efficient data-movement for lightweight I/O," in *Proceedings of the 2006 International Workshop on High Performance I/O Techniques and Development of Very Large Scale I/O Systems*. IEEE, 2006.

[12] C. Ulmer, S. Mukherjee, G. Templet, S. Levy, J. Lofstead, P. Widener, T. Kordenbrock, and M. Lawson, "Faodel: Data management for next-generation application workflows," in *Proceedings of the 9th Workshop on Scientific Cloud Computing*, ser. ScienceCloud'18. ACM, 2018, pp. 8:1–8:6.

[13] "Faodel Software Repository," https://github.com/faodel/faodel, 2019, accessed: 12-April-2019.

[14] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014.

[15] Los Alamos National Laboratory, "Trinity," https://www.lanl.gov/projects/trinity/, retrieved 12 April 2019.

[16] Top500.org, "November 2018 — top500 supercomputer sites," https://www.top500.org/lists/2018/11/, 2018, accessed: 02-April-2019.

[17] T. Trader, "US leads supercomputing with #1, #2 systems and petascale arm," *HPCwire*, 2018.

[18] P. Shivam, P. Wyckoff, and D. Panda, "EMP: zero-copy OS-bypass NIC-driven gigabit ethernet message passing," in *SC'01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. IEEE, 2001, pp. 49–49.

[19] D. E. Bernholdt, S. Boehm, G. Bosilca, M. Gorentla Venkata, R. E. Grant, T. Naughton, H. P. Pritchard, M. Schulz, and G. R. Vallee, "A survey of MPI usage in the US exascale computing project," *Concurrency and Computation: Practice and Experience*, p. e4851, 2017.

[20] D. Li, K. W. Cameron, D. S. Nikolopoulos, B. R. De Supinski, and M. Schulz, "Scalable memory registration for high performance networks using helper threads," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*. ACM, 2011, p. 38.

[21] T. S. Woodall, G. M. Shipman, G. Bosilca, R. L. Graham, and A. B. Maccabe, "High performance RDMA protocols in HPC," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2006, pp. 76–85.

[22] C. Jia, J. Liu, X. Jin, H. Lin, H. An, W. Han, Z. Wu, and M. Chi, "Improving the performance of distributed TensorFlow with RDMA," *International Journal of Parallel Programming*, pp. 1–12, 2017.

[23] M. Li, K. Hamidouche, X. Lu, H. Subramoni, J. Zhang, and D. K. Panda, "Designing mpi library with on-demand paging (odp) of infiniband: challenges and benefits," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 433–443.

[24] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker, "Revisiting network support for RDMA," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM'18, 2018, pp. 313–326.

[25] InfiniBand Trade Association, *InfiniBand Architecture Specification, Volume 1, Release 1.2.1, Annex A17: RoCEv2*, September 2014.