Evaluating the semantic match between OpenSHMEM and OFI

Fredrik Pe Ingebrigtsen
Norwegian University of Science and Technology,
fredripi@stud.ntnu.no

***Abstract*—Communication overhead is an important factor in HPC. As HPC systems continues to grow, good scalability is crucial and to attain this overheads should be minimized. A good semantic match between layers, i.e, a tight functional mapping from one layer to another, helps the development and maintaining of good performance as mismatches may lead to extra overheads. This is particularly important in the network interface, which translates communication abstractions from several different programming models to different underlying hardware interconnects.**

OFI is an open source network interface, focusing on application needs and scalability, that aims to create a tight semantic match between application middleware, such as MPI, SHMEM, PGAS, and the underlying networks. We evaluate the semantic match between OFI and OpenSHMEM, measuring the overhead latency of our OpenSHMEM implementation. The results are encouraging, with an overhead of 7 microseconds for larger messages — close to zero in percentage of total size. We suggest further benchmarking to make a comparison with alternative methods.

I. Introduction

Modern HPC systems have a hierarchical layout, where shared-memory compute nodes with multiple processors are connected via high-speed networks. Speed and computing power are essential qualities, and effective communication is a necessary component. Any additional overhead in the network will reduce performance and limit scalability. Hence it is important for the network interface to offer a good translation between the network capabilities and the abstractions used by an application. This mapping we call the semantic match.

The semantic match between two software layers is a measure of how well functions and data structures map from one to the other. Having a good semantic match helps development and maintenance of optimized software, reducing the effort demanded of the programmer. As there are several different parallel programming models in use in HPC, which often differ semantically in how they perform communication, and a multitude of different architectures, there exists some duplicated optimization efforts.

Open Fabrics Interfaces (OFI) is an open-source collaborative effort to provide efficient network communication services for parallel programs [7]. The core component is the `libfabric` library which defines and exports the user-space API of OFI. The key design goal of OFI is to provide a good semantic match between different programming models, e.g., MPI, OpenSHMEM, and multiple underlying communication services, increasing the portability of optimized software.

In this paper, we evaluate the semantic match between OFI and Open Symmetrical Hierarchical MEMory (OpenSHMEM) with the aim of identifying strengths and potential weaknesses. Concretely, we implement key functions, specifically `shmem_put` and `shmem_get`, from the OpenSHMEM Specification with OFI over InfiniBand, and quantify the overhead in terms of latency of `shmem_put` (`shmem_get` was only tested for correctness due to time constraints).

II. Background

Figure 1 shows a model of the most important layers in a parallel programming environment. Several parallel programming models have been developed to effectively implement parallel programs in different HPC systems. These are usually optimized together with the network interface for each underlying network provider. The network interface provides a bridge between applications and hardware, where a good match between programming abstractions to network capabilities is crucial.

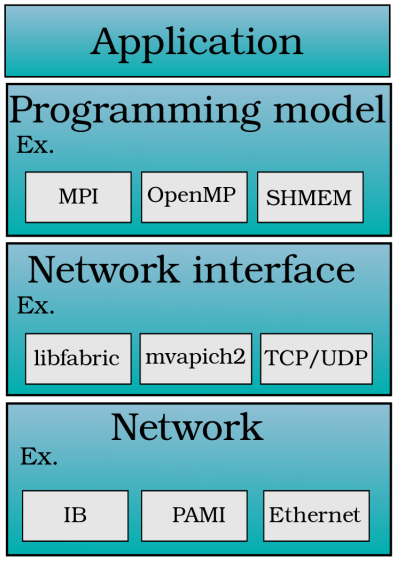

Fig. 1. Different layers in an HPC system.

A. Parallel Programming Models

A parallel programming model describes an abstract parallel machine, with which one can express algorithms and their composition in programs [10]. It also contains a memory

model that describes how and when memory accesses can become visible to the different parts of a the computer. A parallel programming model is often associated with one or several parallel programming languages or libraries that realize the model. Programming models increases portability of parallel algorithms across a wider range of parallel programming languages and systems. We give a brief survey of widely used parallel programming models, and demonstrate some simple Hello World programs in C using MPI, OpenMP and OpenSHMEM.

1) Message Passing - Distributed Memory: In the distributed memory model, several RAMs (random access machines) run asynchronously and communicate with messages sent over a network. Messages can be blocking or non-blocking, and there are also collective messaging operations such as broadcast, gather and reduce. The message passing model is a widely used computation model in HPC today, in part because of the control it gives the user in terms of scheduling, buffering of data, and computation communication overlapping, though this makes programs more prone to errors and harder to understand and maintain.

The dominant message passing interface is MPI, an interface, which is implemented and optimized for almost every distributed memory architecture, and therefore offers a great portability and speed [14]. Because distributed memory programs are quite easily executed on shared memory computers (the reverse being more difficult) MPI is also regularly used as such. Furthermore, newer MPI standards (MPI-2/3) implement additional features like one-sided communication and fork-join style execution, to accommodate the PGAS model (Partitioned Global Address Space).

MPI

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  int rank, size;

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &size);

  printf( "Hello World from process %d \
    of %d\n", rank, size );

  MPI_Finalize();
  return 0;
}
```

Listing 1: Hello World using MPI in C.

2) Shared Memory: A shared memory machine consists of different threads of execution that have access to a shared memory region. The threads run asynchronously and conflicting memory reads/writes must be handled by the system or the programmer. Shared memory model is the prominent model for small scale systems, notably SMP (symmetric multiprocessing) systems. Since large-scale systems have recently begun consisting of clusters of SMP nodes, a combination of shared memory on the nodes with message passing in between can be used [11].

OpenMP is an API that supports shared-memory programming in C, C++, and Fortran. It combines Single Program Multiple Data (SPMD) and fork-join styles of execution and offers work sharing constructs that allow distribution of loop iterations among threads [15].

OpenMP

```c
#include <omp.h>
#include <stdio.h>

int main (int argc, char **argv)
{
  int nts, tid;

#pragma omp parallel private(nts, tid)
  {
    tid = omp_get_thread_num();
    nts = omp_get_num_threads();
    printf( "Hello World from thread %d \
      of %d\n", tid, nts);
  }
  return 0;
}
```

Listing 2: Hello World using OpenMP in C.

3) Partitioned Global Address Space (PGAS): The PGAS model provides a shared memory abstraction on distributed memory machines. It exposes the underlying distributed memory organization to the programmer to give more control over the placement of data structures. Specifically, it allows portions of the shared memory to have affinity for a process, exploiting the locality of reference principle. PGAS tries to combine the advantages of a SPMD style for distributed systems (as employed by MPI) with the data referencing semantics of shared memory systems [12]. The main advantage of having a shared memory abstraction is easier handling of irregular communication and data structures between processes.

Implementations of the PGAS model are either languages, such as Unified Parallel C (UPC), Coarray Fortran (CAF), Chapel, and X10 (the last two currently under development), or libraries like OpenSHMEM and UPC++.

OpenSHMEM is an open source standard specification and reference implementation of the SHMEM libraries. SHMEM (Symmetrical Hierarchical MEMory) is a family of libraries, providing remote memory access and one-sided communication for shared memory computers, and was later expanded as an implementation of the PGAS model for distributed machines [16].

Some of the currently available OpenSHMEM implementations include: Sandia OpenSHMEM, Mellanox ScalableSHMEM, and experimental implementations in OpenMPI and MPI-3 [13].

B. Network Interface

In this paper we refer to the network interface as the API or interface that exports the network capabilities higher up, and is often created together with a specific network. There are sometimes different layers within the network interface, and they are sometimes not easily separated from implementations

OpenSHMEM

```c
#include <shmem.h>
#include <stdio.h>

int main (int argc, char **argv)
{
  int me, npes;

  shmem_init();
  me = shmem_my_pe();
  npes = shmem_num_pes();

  printf("Hello World from PE %d \
    of %\n", me, npes);

  return 0;
}
```

Listing 3: Hello World using OpenSHMEM in C.

of programming models, such as MPI. OFI is somewhat of a middle layer, porting functionalities to multiple networks via multiple network API's. Following is a summary of some of the most used interfaces.

1) MVAPICH2-X: MVAPICH2 is an implementation of the MPI-3 standard on different underlying transport interfaces, such as InfiniBand, iWARP, RoCE, and Omni-Path. MVAPICH2-X is an extension that offers a hybrid programming model, also supporting PGAS in the form of OpenSHMEM, UPC/UPC++ and Coarray Fortran (CAF). It is widely believed that a hybrid of MPI and a PGAS language is fitting for many scientific computing problems, especially for exascale computing [5]. Currently, MVAPICH2-X is a binary only release.

2) Mellanox MXM, ScalableSHMEM: ScalableSHMEM and MXM have been co-designed to overcome scalability issues to improve efficiency. In [9], the approach of co-designing communication libraries with the underlying hardware interconnect is presented.

3) Sandia Portals 4: The Portals Network Programming Interface, from Sandia National Laboratories, is designed to operate on scales ranging from a small number of commodity desktops connected via Ethernet to massively parallel systems connected with different networks [6]. Portals 4 has been under development since 2007 and was enhanced to better suit the PGAS model, currently being supported by Sandia OpenSHMEM, GASNet, UPC and others.

4) IBM PAMI: The IBM Parallel Active Messaging Interface is a messaging interface used in IBM HPC platforms, first developed for the Blue Gene systems [8]. It is also extensible to other networks, such as InfiniBand. It supports a broad range of programming models, such as SHMEM, MPI, etc. and applications can also directly use PAMI. The Asynchronous Partitioned Global Address Space runtime (APGAS) is a library that is specifically used by PGAS languages such as UPC, X10 and CAF in the IBM environment.

5) cray gni/dmapp: The user Generic Network Interface uGNI, and DMAPP are interfaces from Cray, Inc. that provide communication services for the Gemini network. DMAPP, logically shared, distributed memory applications, is specifically designed for the PGAS model.

6) OpenFabrics Interface (OFI): Open Fabrics Interfaces is an open-source collaborative effort to provide efficient fabric communication services for parallel programs. The design goals of OFI is to develop interfaces aligned with application needs, be implementation agnostic, and offer scalability and portability by providing a good semantic match between the application layer and underlying communication services. OFI currently supports several underlying networks including InfiniBand, RoCE and iWARP (all via the verbs API), TCP, UDP, Cray GNI, and Mellanox MXM. Some support are experimental as OFI is very much a work in progress.

III. OpenSHMEM implementation over libfabric

One-sided communication and remote memory access is the main part of OpenSHMEM and as such most functions in the OpenSHMEM specification are different put and get functions. The rest of the functions are for various other tasks including initialization, finalization, synchronization, and communication collectives. We chose to focus on inter-node communication performance and implemented `shmem_put` and `shmem_get` (two general datatype memory access functions), using libfabric. A model of where our implementation lies in the stack is shown in figure 2. This figure is slightly simplified as libfabric uses the verbs API (the InfiniBand API), which can be thought of as a layer between libfabric and the InfiniBand.

The OpenSHMEM processing elements are modeled using libfabric endpoints, which are transport level communication portals containing the resources needed for communication. The PGAS model map well to the reliable datagram messages (RDM) endpoint type [1], which offers connection-less reliable communication. Unfortunately, the verbs RDM support is still experimental and did not work on our system. We instead used the MSG endpoint, with connection-oriented data transfers. This means there a slight extra overhead when transferring the data, which will negatively affect scalability.

Synchronization is done using counters, a light-weight completion mechanism that records completions of request in limited detail. Endpoints are addressed using an address vector which translates well to the logical rank of OpenSHMEM PE's.

a) `shmem_put`: is a one-sided communication method for copying data from a contiguous local data object to a data object on a specified PE. This is a remote memory access routine where libfabric's own RMA API, specifically `fi_write`, is a good match. The data transfers are done to memory regions associated to the endpoints.

b) `shmem_get`: is like shmem_put only it copies the data *from* a specified PE *to* a data object at the calling PE. `fi_read` provides a near one to one match.

The code for the `shmem_put/get` functions of the implementation is included in the appendix, highlighting the close semantic match with OFI functions.

IV. Method

The experiments was done at a HP BL 460c Gen8 compute cluster at the University of Tromsoe [4]. Two nodes, each

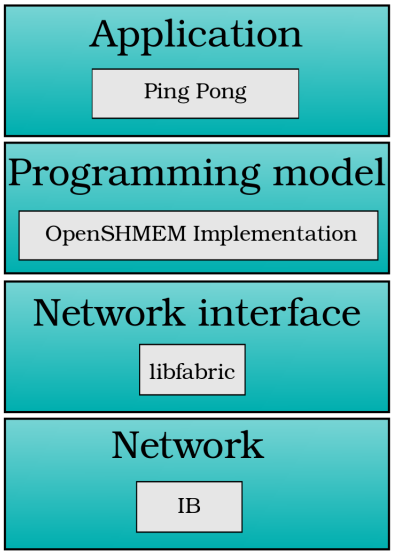

Fig. 2. OpenSHMEM implementation in the stack.

with two 8 core 2.6GHz Intel Sandy Bridge CPUs and 32GB memory, was used for the message-passing. They share an InfiniBand (MT26428) interconnect, and are running CentOS Linux 6.7 with kernel version 2.6.32-431.23.3.el6.x86 64. We used libfabric v1.1, and GCC v4.4.7 with the -O2 option.

We created a simple ping pong implementation for the experiments. The program is largely based of fabtests, a set of examples for libfabric. Firstly the client and server sets up necessary fabric resources, used to handle different requests and synchronization. Afterwards the client issues a connection request to the listening server, and begins message transfer when a connection is established. The time measurements is done only for the actual message sending, not including any initialization of resources. The results taken from 1000 iterations.

V. Results

In this section we present the results of our OpenSHMEM implementation in a small ping pong benchmark. We measure the semantic match by quantifying the overhead in microseconds for our OpenSHMEM implementation compared to using libfabric directly. We also tested direct libfabric performance on our IB network using the verbs API vs. TCP.

The verbs vs. sockets results are a plot of the average of all the measurements, while the overhead of the OpenSHMEM implementation are plotted as box plots. The box indicates the 25 % and 75 % percentile, known as the IQR (interquartile-range), and whiskers at the lowest observation inside 1.5 * IQR. The middle bar is the median, and outliers are shown as additional marks. We removed outliers outside the interval $[x_{25\%} - 1.5(x_{75\%} - x_{25\%}), x_{75\%} + 1.5(x_{75\%} - x_{25\%})]$.

A. Ping Pong using Verbs and Sockets

As an experiment to ensure our libfabric setup was configured correctly, we tested a simple ping pong program, communicating using verbs or sockets over InfiniBand. It is worth noting that the sockets provider in libfabric was implemented for portability reasons and not for performance. Figure 3 shows the transfer latency for different message sizes. It uses the `FI_EP_MSG` endpoint type to send messages using the `fi_send` and `fi_recv` libfabric functions.

We observe that communicating over sockets has a larger initial overhead which scales linearly with increasing message sizes. The first inclines in the lines are where the maximum package sizes for Ethernet and InfiniBand are met, at 1500 and 4000 bytes respectively.

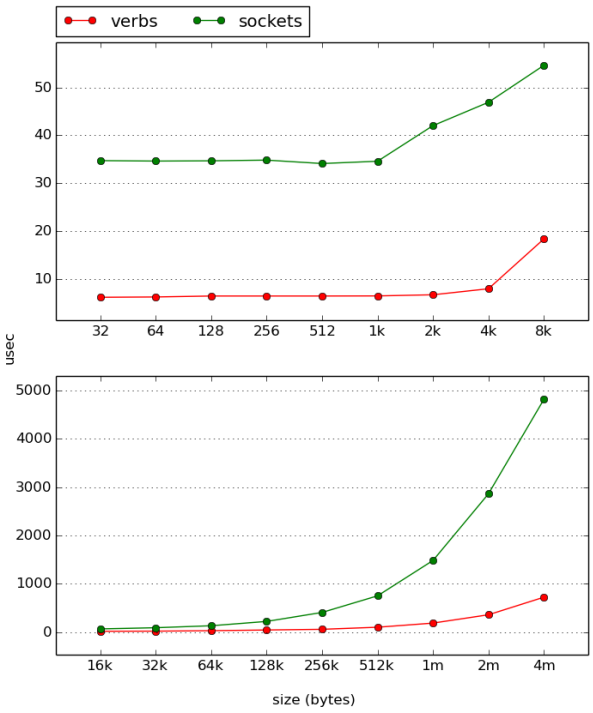

Fig. 3. Microseconds per transfer size.

B. SHMEM Put Implementation

For performance testing the OpenSHMEM functions two PE's are set up on two nodes, where both write to a memory buffer on the other processing element (node) using remote memory access.

This benchmark measures the overhead of our implementation of a `shmem_put` call for different message sizes, as the difference in runtime against the same message transfer using the libfabric `fi_write` call directly. A substantial amount of the data in figure 4 was outliers, between 10 - 30 % for some of the larger sizes, reaching as far as a hundred microseconds. These were removed from the boxplots for better visuals. Figure 7 shows the distribution of overheads in percentage of total time for message size 1 MiB. Reasons for these outliers appears to be noise, potential sources may come from the system, such as network background traffic or interrupts. For smaller message sizes the overhead is close to zero, and later on stabilizes around 7 microseconds.

In figure 6 the overhead is plotted as percentage of total time. We observe the overhead percentage decreasing as message sizes increases, to almost negligible heights for 4 MiB messages — an indication of good scalability.

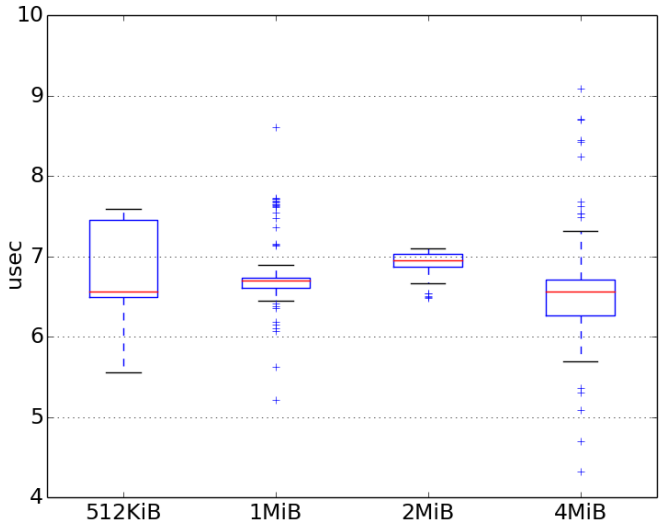

Fig. 5. `shmem_put` overhead for larger messages.

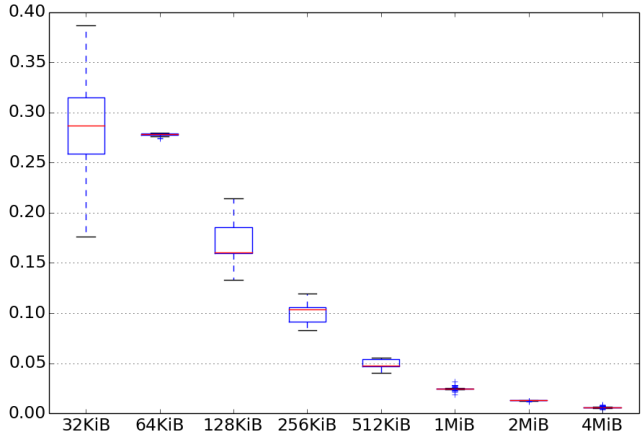

Fig. 6. `shmem_put` overhead percentage of total time.

VI. Discussion

The amount of variance detected in our measurements are large enough to question the results' validity. A brief discussion of our methods and what could have been done differently follows.

The analysis of overhead latency was made by taking two separate measurements, one with the `shmem_put` implementation and one without, then evaluating the difference between each single measurement. These measurements are independent of each other, meaning that any variance detected in the system will more or less double, knowing that the variance of the difference between two random variables equals the sum of the variances minus the co-variance (this still holds for skewed distributions). So to minimize the impact of noise these measurements could possibly have been carried out differently.

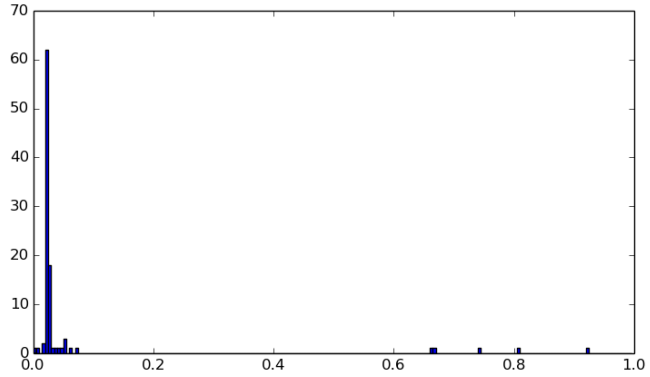

Fig. 7. Distribution of overhead for message size 1MiB.

VII. Conclusion and Further Work

In this paper we have looked at the semantic match between OFI and OpenSHMEM. We give some background information on parallel programming, and provide an overview of existing alternative paths for parallel programming on clusters. Our experiments show that the overhead of the OpenSHMEM functions are low, around 7 microseconds for the larger message sizes.

As further work, we would like to implement a larger subset of the OpenSHMEM specification, enough to implement one or more benchmarks, and subsequently perform a comparison of our OpenSHMEM – OFI stack to other alternatives. Measuring the instruction count, as opposed to latency, may give a more accurate representation of the semantic match.

Appendix
OpenSHMEM implementation

Listing 4 is an excerpt from our OpenSHMEM implementation showing only the isolated `shmem_put` and `shmem_get` functions.

Appendix
Problem Description

Open Fabrics Interfaces (OFI) is an open-source framework that provide efficient fabric communication services for parallel programs. The core component is the libfabric library which defines and exports the user-space API of OFI. The key design goal of OFI is to provide a good semantic match between the application and the underlying communication services. The lack of a good semantic match will result in an undesirable translation overhead when interfacing the application and the communication services. To achieve good scalability, it is critical that the communication overhead is as low as possible.

The goal of this project assignment is to study OFI with the aim of identifying strengths and potential weaknesses. Concretely, the student should conduct a literature study that compare OFI to related communication fabric APIs. If time

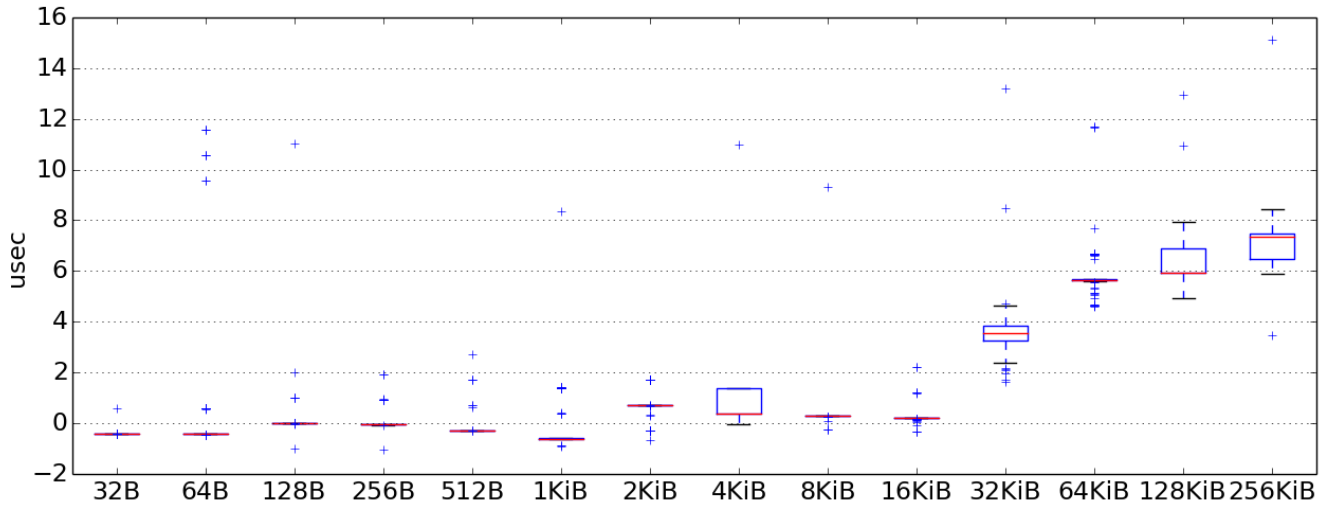

Fig. 4. `shmem_put` overhead in microseconds. The most distant outliers were removed.

shmem put and get

```
int shmem_put(size_t len, const void *buffer)
{
  int ret = fi_write(ep, buffer, len,
    fi_mr_desc(mr), 0, remote.addr,
    remote.key, ep);

  ret = ft_get_tx_comp(++tx_seq);
  if (ret)
    return ret;
}

int shmem_get(size_t len, const void *buffer)
{
  int ret = fi_read(ep, buffer, len,
    fi_mr_desc(mr), 0, remote.addr,
    remote.key, ep);

  ret = ft_get_tx_comp(++tx_seq);
  if (ret)
    return ret;
}
```

Listing 4: The `shmem_put` and `shmem_get` functions from our OpenSHMEM implementation.

permits, the student should quantify the overhead (in terms of number of instructions) of implementing key functions from the Open Symmetrical Hierarchical MEMory access (OpenSHMEM) API with OFI.

References

[1] Miao Luo, Kayla Seager, Karthik S Murthy, Charles J Archer, Sayantan Sur, and Sean Hefty. Early evaluation of scalable fabric interface for pgas programming models. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, page 1. ACM, 2014.

[2] Paul Grun, Sean Hefty, Sayantan Sur, David Goodell, Robert D Russell, Howard Pritchard, and Jeffrey M Squyres. A brief introduction to the openfabrics interfaces-a new network api for maximizing high performance application efficiency. In *High-Performance Interconnects (HOTI), 2015 IEEE 23rd Annual Symposium on*, pages 34–39. IEEE, 2015.

[3] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, page 2. ACM, 2010.

[4] Stallo documentation. https://www.sigma2.no/content/stallo. Accessed: 2016-03-17.

[5] MVAPICH2 documentation. http://mvapich.cse.ohio-state.edu/overview/. Accessed: 2016-04-06.

[6] Portals4 documentation. http://www.cs.sandia.gov/Portals/portals4.html. Accessed: 2016-04-06.

[7] libfabric webpage. https://ofiwg.github.io/libfabric/. Accessed: 2016-06-13.

[8] Sameer Kumar, Amith R Mamidala, Daniel A Faraj, Brian Smith, Michael Blocksome, Bob Cernohous, Douglas Miller, Jeff Parker, Joseph Ratterman, Philip Heidelberger, et al. Pami: A parallel active message interface for the blue gene/q supercomputer. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 763–773. IEEE, 2012.

[9] Gilad Shainer, Todd Wilde, Pak Lui, Tong Liu, Michael Kagan, Mike Dubman, Yiftah Shahar, Richard Graham, Pavel Shamis, and Steve Poole. The co-design architecture for exascale systems, a novel approach for scalable designs. *Computer Science-Research and Development*, 28(2-3):119–125, 2013.

[10] Christoph Kessler and Jrg Keller. Models for parallel computing: Review and perspectives. In *PROCEEDINGS, PARS*, pages 13–29, 2007.

[11] David S Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, page 10. IEEE Computer Society, 2000.

[12] Filip Blagojević, Paul Hargrove, Costin Iancu, and Katherine Yelick. Hybrid pgas runtime support for multicore nodes. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, page 3. ACM, 2010.

[13] Jithin Jose, Jie Zhang, Akshay Venkatesh, Sreeram Potluri, and Dhabaleswar K DK Panda. A comprehensive performance evaluation of openshmem libraries on infiniband clusters. In *Openshmem and Related Technologies. Experiences, Implementations, and Tools*, pages 14–28. Springer, 2014.

[14] MPI website. https://www.mpi-forum.org/. Accessed: 2016-05-10.

[15] OpenMP website. http://openmp.org/wp/. Accessed: 2016-05-10.

[16] OpenSHMEM website. http://openshmem.org/site/. Accessed: 2016-05-10.