Multi-Threaded OpenSHMEM: A Bad Idea?

Gabriele Jost
Intel Corporation
3600 Juliette Lane
Santa Clara, CA 95054, United States
+1 408 653 9264
Gabriele.Jost@intel.com

Ulf R. Hanebutte
Intel Corporation
705 5th Ave S Suite 500
Seattle, WA 98104, United States
+1 206 701 8745
Ulf.R.Hanebutte@intel.com

James Dinan
Intel Corporation
75 Reed Road
Hudson, MA 1749, United States
+1 978 553 1216
James.Dinan@intel.com

ABSTRACT
The purpose of this document is to stimulate discussions on support for multi-threaded execution in OpenSHMEM. Why is there a need for any thread support at all for an API that follows a shared global address space paradigm? In our ongoing work, we investigate opportunities and challenges introduced through multi-threading, namely implementation challenges and opportunities and required – as well desirable – extensions to the API.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *Concurrent programming structures, Patterns, Frameworks*

General Terms
Algorithms, Performance, Languages.

Keywords
OpenSHMEM, threading, hybrid programming.

1. MOTIVATION
Current large-scale HPC systems are typically clusters of multi-core nodes. The trend in hardware architectures suggests that the number of cores per node will continue to increase while memory per core decreases. To take advantage of these cores, the number of threads and processes executed per node will also increase. However, processes with separate virtual address spaces partition the node memory, resulting in memory pressure. Hybrid parallel programming with processes and threads is a popular remedy to this problem, because it extends existing process-based parallel programming libraries with intranode data sharing [3].

The OpenSHMEM parallel programming model follows the partitioned global address space (PGAS) paradigm, a model that is well suited for platforms with non-uniform memory access latencies. The OpenSHMEM specification also provides support for direct load/store memory access within a node, which alleviates the need to use threads for this purpose.

Why should we introduce threads to OpenSHMEM if inter-PE shared memory can be already exploited directly through the OpenSHMEM interface? Doesn't this just introduce overhead through e.g. forking, joining and other thread management issues? Hybrid programming with threads also introduces extra application development effort and program complexity. Nevertheless, lessons learned from other programming models indicate that there might be other benefits provided by a hybrid approach.

We discuss some preliminary observations regarding the OpenSHMEM model and challenges when programming for SMP node clusters in Section II. In Section III we give examples for opportunities using a hybrid approach and in Section IV we discuss some ideas on necessary support within OpenSHMEM. In Section V we summarize the status of our current ongoing work and outline future work. Through this work, we aim to stimulate discussion between application developers, the OpenSHMEM specification committee, and OpenSHMEM implementers to determine requirements to extend OpenSHMEM for future extreme-scale environments.

2. SHMEM ON SMP NODE CLUSTERS

2.1 Preliminary Observations
OpenSHMEM is a library API that provides a single program multiple data (SPMD) execution, in which participating processes (the places where work occurs are called Processing Elements or PEs) view each other's data through a partitioned global address space (PGAS). Note that in a multi-threaded environment, the PE corresponds to the process, not the thread. The OpenSHMEM model defines two shared, symmetric memory segments: data and heap segments. The data segment contains statically declared shared objects, and the heap contains dynamically allocated shared objects.

Symmetric objects are accessed using a PE's local pointer to the given symmetric object, and the desired target PE. When running within the same SMP node, shared data segments of peer PEs can be mapped into the address space of a given PE. Support for this is provided through the shmem_ptr function, which queries the local pointer for another PE's instance of a symmetric object.

2.2 Performance Challenges on SMP Node Clusters
HPC systems have become increasingly "non-isotropic" at the node level with a rich hierarchy of shared caches, ccNUMA domains, multiple sockets and a large number of cores. Applications also expose hierarchies in their parallel

decomposition. Such hierarchies are introduced, e.g. by domain decomposition and intra-domain solver routines. Mapping the application sub-domains onto the hardware hierarchy is a challenging problem. For high overall application performance, one must address both communication and computational performance.

Communication performance is impacted by multicore and multisocket anistotropy effects, such as differences in inter-node vs intra-node communication, communication patterns of the application, bandwidth bottlenecks, and impact of shared caches. Placing multiple processes on a node will often yield unnecessary calls to the communication library. A small number of PEs per node may not be able to saturate the network bandwidth and may waste compute resources.

Computational performance is impacted by factors like ccNUMA locality effects, penalties for access across NUMA domain boundaries, messaging pattern and resource requirements.

When employing multiple threads per process, it is often difficult to determine the right balance of processes and threads per node to achieve the best core performance and memory bandwidth. To highlight this with an anecdotal data point, we run the OpenSHMEM NAS Parallel Benchmark (NPB) Scalar Penta-diagonal solver (SP) implementation developed at the University of Houston [11] on a Cray XC30 (16 cores per node + hyper-threading). We used the Cray compiler Version 8.2.5 and Cray SHMEM 6.3.0. We have gathered some execution characteristics that are shown in Figure 1. The goal was not an in-depth analysis of the algorithms nor a performance assessment of the Cray system. We ran two experiments, employing 4 and 16 PES per node. We have normalized the values to a range of 0 to 1 in order to observe how they vary with respect to each other. Increasing the number of PES will result in more, yet shorter messages. The memory requirements increase with growing number of PEs. Most notable is the difference in Mops between 16 and 4 PEs per node. It points to the fact that packing all cores with processes may not always be the most efficient way to execute or may not be possible due to memory limitations. However, reducing the number of PEs leaves a quarter of the cores idle. While this example is rather trivial, we have noted similar effects in real-world cases [5].

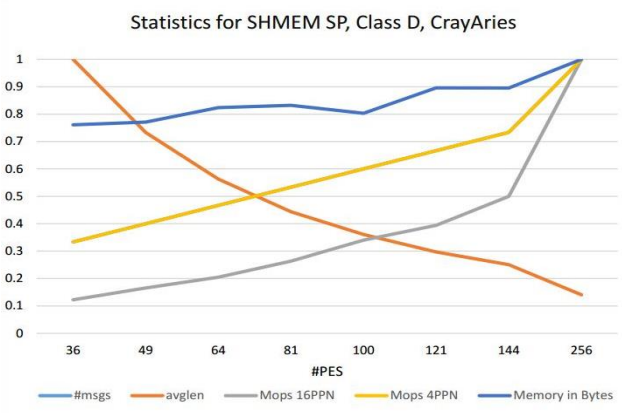

Figure 1. Characteristics of OpenSHMEM NPB SP.

An efficient OpenSHMEM implementation can avoid unnecessary communication calls within a node. This is demonstrated in the optimized OpenSHMEM implementation by [7] and by OSHMPI [4], an OpenSHMEM implementation based on MPI. It will, for example, not issue calls to MPI_Put within a node but rather use direct load/store provided by MPI-3. Similarly, an OpenSHMEM implementation could address some of the ccNUMA aspects with respect to avoiding inter-domain contention through efficient placement of symmetric data and communication buffers.

So, is hybrid OpenSHMEM+Threads indeed bad idea? In the next section we will look into how hybrid programming could improve application performance.

3. OPPORTUNTIES AND BENEFITS OF HYBRID PROGRAMMING

Applications that utilize a hybrid execution model can be categorized as shown in Figure 2.

The "master-only" (PE only) mode typically has calls to the communication API only outside of multithreaded regions. This requires the lowest level of thread-safety in which only a master thread issues communication calls. Compute threads typically sleep during communication calls and it may be hard to sustain full communication bandwidth with just a single process per node.

A higher level of thread safety is required if calls to the communication API occur within parallel regions. Executing in this mode provides the opportunity to overlap communication and computation by assigning a subset of threads to communication and the rest to computation. It also enables pipelined thread execution across multiple processes and facilitates the implementation of wave-front methods.

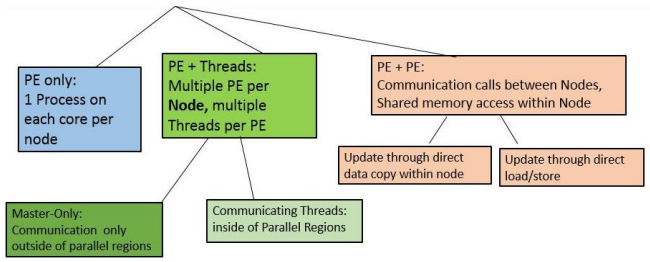

Figure 2. Hybrid Communication API Usage Models.

Most hybrid programming today is based on employing two-sided MPI communication and OpenMP for multithreading. Users have observed considerable benefits when exploiting multi-level parallelism based on domain composition [3].

Certain classes of applications can benefit by employing the master-only mode. Examples for this are the multi-zone versions of the NAS Parallel Benchmarks [12]. These benchmarks capture the behavior of applications from the field of Computational Fluid Dynamics. Complex structural domains are covered by a fixed number of computational grids, referred to as zones. Computations within each zone require only the occasional update of boundary values. This lends itself to coarse-grained parallelism based on message passing, assigning a number of zones to each processing element. Fine-grained parallelism can be exploited via multithreading in the solver routines within each zone. The number of grid points within each zone often varies significantly.

We summarize the following benefits from hybrid programming, which are discussed in [3]:

Flexible load balancing: A smaller number of PEs often allows room to distribute work more evenly among PEs. In addition, PEs with a very high workload could use multiple threads. Dynamic thread scheduling can improve load-balancing on thread level.

Increase exposed parallelism: In an application where the outer level of parallelism is limited, multithreading can provide a convenient way to expose additional fine-grained, multi-level parallelism. An example for this is LU-MZ from the NPB-MZ suite.

Lower memory requirements: Multithreading can reduce the amount of replicated data, the size of the symmetric heap, and the size of required communication buffers. In general, one can estimate the required memory per PE as the sum of: static data segment, private heap, symmetric heap, stack, and communication buffers. This will be extremely important for addressing memory pressure on systems with thousands of cores per node. In particular, the inter-PE shared memory model cannot address the memory consumed per-node by static data.

Others: The hybrid approach may offer a convenient way to exploit task-based parallelism, for example the OpenMP 4.0 task extensions or OmpSs [1], [14]. An example is presented in [6]. MPI and OpenMP tasking was successfully employed to increase parallelism by overlapping computational work with independent MPI global communication.

An example that requires a higher level of thread-safety is the SNAP Application Proxy [13]. It solves the linear Boltzmann transport equation, i.e. determining the number of neutral particles. A nested iterative algorithm is applied to solve the "flux" function of seven independent variables at each time step: a 3-D spatial mesh, angles of travel and energy groups. The parallelization strategy is based on an outer loop over all energy groups implemented with multithreading and task scheduling for load balancing. Each energy group is solved in parallel applying a wavefront method that utilizes message passing on a two-dimensional domain decomposition. This requires the threads to issue communication calls. Furthermore, for the MPI implementation of this application, a higher thread support level correlates to higher parallel efficiency of the algorithm.

3.1 Analysis of Hybrid Programming in a One-Sided Context

As we have discussed, the use of hybrid programming is commonplace in message passing applications. The idea of using hybrid programming in the context of one-sided communication and multi-threading is not new. In the late 1990's the shared memory Multi-Level Parallelism (MLP) technique ([8], [9]), developed at NASA Ames was shown to be successful in improving the performance of CFD codes on large-scale shared memory machines.

To achieve extreme-scale performance, we anticipate a need for exploiting parallelism at multiple levels, taking into account all system hierarchies, including memory hierarchies as well as hierarchies in the system interconnect. While this can be achieved through processes with interprocess shared memory, a hybrid programming model may offer a better path to mapping application multi-level parallelism onto system hierarchies. While OpenSHMEM with interprocess shared memory addresses two levels in the hierarchy, OpenSHMEM plus threads offers additional capabilities for finer mapping to the system hierarchy. However, efficient support for threads must be provided by both the API and runtime system implementation. How much should be exposed to the user via the API, and how much hidden under the hood? The following Section IV will discuss some issues.

4. SOME THOUGHTS ON THREAD SUPPORT

A basic requirement for threading support in OpenSHMEM is thread safety. The OpenSHMEM community has already started to propose and implement thread-safe OpenSHMEM. See for example the work by [10], which defines several levels of thread safety, similar to MPI. Library-level thread safety is straightforward and well understood. However, it has the disadvantage that thread synchronization is conjoined because threads share the communication state of a single process. Calls to synchronization routines like shmem_fence, shmem_quiet and shmem_barrier affect all threads and may introduce unnecessary and unwanted synchronization points that change the execution flow and introduce inefficiencies. Furthermore, the threads share network-level communication resources and state. For this reason, ten Bruggencate et al. [10] also introduce an API that binds threads to the SHMEM runtime, allowing them to communicate independently.

Another approach is to declare each thread within a node to be an OpenSHMEM PE, noting that PEs within a node naturally share their memory. A disadvantage is the interoperability with other programming models, for example OpenMP, OmpSs or Cilk. In addition, symmetric objects are required for all PEs and this model introduces a need for symmetric data segments in a single virtual address space.

Ideally, we would like to have both – PEs as well as threads (or some other lightweight execution units). Memory resources should depend on the number of PES only, while the threads should have their own communication resources. Threads should be able to synchronize individually. Recently, a new approach to thread integration was proposed that introduces per-thread communication contexts [2]. Contexts allow threads to produce individual streams of communication operations, without binding them to the SHMEM runtime, eliminating interference between threads while providing interoperability with a variety of threading models.

While many important lessons can be learned from past hybrid programs, the characteristics of new OpenSHMEM applications may differ considerably from those based on domain decomposition and coarse-grained communication. We foresee the need for a high bandwidth and low latency for small put, get, and atomic memory update operations. Hybrid OpenSHMEM models must also aim to support these requirements, which have additional sensitivities to threading overheads.

5. CONCLUDING REMARKS

OpenSHMEM+Threads provides performance and capability opportunities beyond those of processes with interprocess shared memory access, including flexible solutions to memory pressure, multi-level parallelism, and optimized node-level programming. Through such mechanisms, hybrid parallel programming with OpenSHMEM may have the potential to significantly increase application performance on extreme scale systems.

We discussed two major opportunities for utilizing a hybrid model. Firstly, memory requirements may pose a major obstacle for OpenSHMEM, due to its symmetric memory model. By adding a second, non-symmetric node-level programming model, users can adjust the level of per-node symmetry. Secondly, we envision that a hierarchical programming paradigm is better suited to exploit application-inherent multi-level parallelism, thereby providing better mapping of the application onto system hierarchies.

Recent activities within the OpenSHMEM community aim to extend the standard to introduce hierarchies, teams, and communication contexts. These extensions are potential avenues for providing the necessary constructs to the application developer to develop applications that can take advantage of all the compute power that future high core-count and high node-count systems will provide. In our future work, we plan to more deeply explore these issues by evaluating inter-PE shared memory extension to OpenSHMEM and comparing their effectiveness with OpenSHMEM+Threads with respect to both performance as well as programmability.

6. REFERENCES

[1] Barcelona Supercomputer Center (BSC), "The OmpSs Programming Model", Online: https://pm.bsc.es/ompss. Accessed Sep. 2014.

[2] Dinan, J. and Flajslik, M.. "Contexts: A Mechanism for High Throughput Communication in OpenSHMEM.", In Proc. 8th Intl. Conf. on Partitioned Global Address Space Programming Models (PGAS). Oct. 7-10, 2014.

[3] Rabenseifner, R.; Hager, G.; Jost, G., "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes," In Proc. 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP). Feb. 2009.

[4] Hammond, J., R.; Ghosh, S.; and Chapman, E., M., "Implementing OpenSHMEM using MPI-3 one-sided communication.", OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools. Lecture Notes in Computer Science, Volume 8356, pp 44-58. 2014.

[5] Jost, G. and Robins, R., "Experiences using hybrid MPI / OpenMP in the real world: Parallelization of a 3D CFD solver for multi-core node clusters," Scientific Programming, IOS Press, Special Issue on Exploring Languages for Expressing Medium to Massive On-Chip Parallelism, pp. 127 – 138, IOS Press, Jan 2011.

[6] Koniges, A. E., et. al., "Application Acceleration on Current and Future Cray Platforms", Proc. 52nd Cray User Group Meeting, Simulation Comes of Age (CUG). Edinburgh, United Kingdom. May, 2010.

[7] Potluri, S.; Kandalla, K.; Bureddy, D.; Li, M.; Panda, D.K.: "Efficient Intranode Desgins for OpenSHMEM on Multicore Clusters". In: 6th Conference on Partitioned Global Address Space (PGAS). 2012.

[8] Taft, J., R. "Performance of the Overflow-MLP CFD Code on the NASA Ames SGI Origin," NAS Technical Report NAS-00-005. March, 2000.

[9] Taft, J.R. "Multi-Level Parallelism (MLP): A Simple Highly Scalable Approach to Parallelism," In Proc. HPCCP/CAS Workshop. 1998.

[10] ten Bruggencate, M.; Roweth, D.; and Oyanagi, S. "Thread-safe SHMEM extensions," OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools. Lecture Notes in Computer Science, Volume 8356, pp 178-185. 2014.

[11] University of Houston, "NAS parallel benchmarks for OpenSHMEM, version 1.0a," Online: http://openshmem.org/site/Downloads/Examples, Accessed: Sep. 2014.

[12] Van der Wijngaart, R. and Jin, H. "NAS Parallel Benchmarks, Multi-Zone Versions," NAS Technical Report NAS-03-010. July, 2003.

[13] Zerr, Robert J. and Baker, Randal S., "SNAP: SN (Discrete Ordinates) Application Proxy," Online: https://github.com/losalamos/SNAP. Accessed: Sep., 2014.