Evaluating data parallelism in C++ using the Parallel Research Kernels

Jeff R. Hammond
jeff.r.hammond@intel.com
Data Center Group
Intel Corporation
Hillsboro, Oregon

Timothy G. Mattson
timothy.g.mattson@intel.com
Parallel Computing Lab
Intel Corporation
Hillsboro, Oregon

ABSTRACT

The Parallel Research Kernels are a set of simple algorithms that correspond to popular classes of high-performance computing applications. We report on their use to evaluate parallel programming models based upon modern C++.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

C++, parallelism, heterogeneity

ACM Reference Format:
Jeff R. Hammond and Timothy G. Mattson. 2019. Evaluating data parallelism in C++ using the Parallel Research Kernels. In *International Workshop on OpenCL (IWOCL'19), May 13–15, 2019, Boston, MA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3318170.3318192

1 INTRODUCTION

The purpose of the Parallel Research Kernels (PRK) [37], is to be a set of application skeletons that exemplify important patterns in parallel processing [25]. Simpler than mini-applications, which are simplified versions of real applications, the PRKs are designed to require no domain science expertise to understand. For example, while there are PRKs inspired by heat equation solvers, distributed Fast Fourier transform (FFT), and discrete ordinates neutron transport, one need not understand any of this to work with the associated code. Each PRK code should be capture one and only one parallel processing pattern, short enough to create or redesign in a single-digit number of days, and be mathematically verifiable regardless of the size of the problem or computing resources used.

In this paper, we report on the development of PRKs associated with a number of parallel C++ frameworks, including the C++17 parallel STL (PSTL) [17], Khronos® SYCL™ [34], Intel® Threading Building Blocks (TBB) [16], Kokkos [8, 9, 22], and RAJA [15, 21]. In order to establish a baseline for evaluating these purely C++ frameworks, we also developed implementations based on OpenMP® 4.5 [29] and OpenCL™ 1.2 [20]. The purpose of this development is to create a set of idiomatic implementations of important parallel processing patterns to enable a fair evaluation of the programming model features and implementation aspects of these frameworks. This paper will describe the similarities and differences revealed by the use of these programming models, as well as some of the implementation details.

Because enabling the programming models research community is one of the primary purposes of the PRK project, the code is available on GitHub [1] with a permissive license (generally BSD-3, but see `COPYING` for details). Additionally, intensive software testing is implemented using Travis CI [2]. In addition, manually testing of most major toolchains is performed regularly on all the platforms to which the developers have access.

2 BACKGROUND AND RELATED WORK

The PRK project includes a number of parallel processing patterns, including nstream, synch_p2p (henceforth referred to as p2p), synch_global, transpose, stencil, dgemm, random, branch, reduce, refcount, and sparse [37]. Not all of these are interesting in every context, of course, and this study focuses on nstream, p2p, stencil and transpose.

The nstream kernel is inspired by STREAM triad benchmark [26, 27], which measures memory bandwidth by performing scaled vector addition equivalent to the BLAS primitive DAXPY. In the context of this study, nstream is primarily useful for understanding the overhead of launching parallel work, as the actual time spent evaluating the kernel is limited by the memory subsystem. There are some exceptions to this, caused by differences in compiler code generation[1], but we do not observe this in our studies, as the Intel compiler is able to generate nontemporal stores in all cases.

The next simplest benchmark of interest is probably transpose, which performs an out-of-place matrix transpose, followed by a trivial update of the input matrix, which helps with validation. The transpose kernel has a similarly low compute intensity as the nstream kernel, but due to its access pattern, stresses the memory hierarchy in ways nstream does not. On modern CPU architectures, transpose will encounter a large number of translation lookaside buffer (TLB) misses when using 4KiB pages. This issue is ameliorated by larger page sizes (e.g. 2MiB and 1GiB) or blocking, which the PRK implementations support.

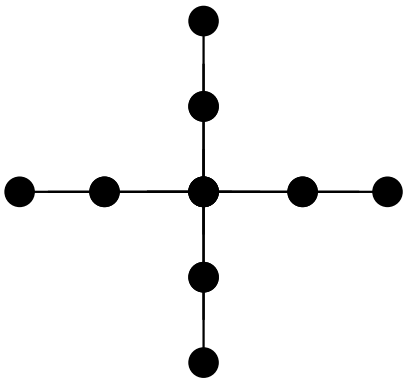
The stencil kernel implements a finite-difference operator corresponding to the heat equation on a two-dimensional grid. The radius of the stencil is configurable – the C++ implementations use a code generation that emits the stencil kernels from radius one to ten, which are invoked using function pointers (where applicable).

[1] See Hager's blog for details [14].

IWOCL'19, May 13–15, 2019, Boston, MA, USA

ACM ISBN 978-1-4503-6230-6/19/05.
https://doi.org/10.1145/3318170.3318192

Figure 1: A pictoral representation of the PRK stencil kernel with the star pattern and radius of two.



The stencil kernel has good spatial locality in its memory access and thus has the potential to be compute bound. All of the C++ implementations support multi-level blocking to improve cache reuse.

The final kernel considered in this study is p2p, which cannot be parallelized in the same way as the aforementioned patterns, due to a data dependency. The p2p pattern is only data-parallel when iterating across anti-diagonals of the matrix, which is a suboptimal memory access pattern. The p2p acts as a relatively strong filter against purely data parallel programming models, which are limited to the hyperplane implementation that iterates over groups of antidiagonals and uses barrier synchronziation between these groups. Programming models that support message-passing, tasks with dependencies, or other forms of point-to-point synchronization can implement the p2p kernel in a more natural way.

We exclude the other kernels from this study for a number of different reasons. In the case of dgemm, the performance difference between a hand-written implementation and an optimized library implementation is so profound that it does not make sense to use it as a study of programming models. A proper evaluation of dgemm could be based on the BLIS template for multithreaded matrix-matrix multiplication [33]; this has already been performed in a limited manner in BLIS [39] and TBLIS [24] by virtue of their support for OS threads and OpenMP, and OpenMP and TBB, respectively.

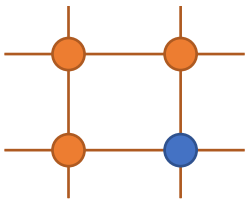
We exclude sparse (sparse matrix-vector product) for a similar reason: the algorithmic nuances of sparse linear algebra are highly dependent on the details of the matrix under consideration and will be largely independent of programming model when implemented properly.

The random kernel is inspired by the HPCC [23] random access benchmark, also known as GUPS. This pattern is designed to saturate the memory subsystem using atomic operations. While it has some programming model dependency with distributed memory [31, 32], the shared-memory behavior is dominated by the performance of atomic operations and unlikely to expose interesting differences between the C++ parallel frameworks we are evaluating. The refcount kernel was designed to stress atomic operations, but should be most sensitive to hardware behavior, rather than programming model.

The synch_global and reduce kernels are primarily of interest in the distributed memory case. Collective operations such as these are still interesting in shared-memory, but a proper evaluation of their support in parallel C++ frameworks will require a new definition of the kernel. Finally, branch was designed to stress branch predictors, rather than software; as a practical matter, the current definition is rather specific to the C89 implementation and would need redefinition to be interesting in the case of parallel C++ frameworks. In particular, a new version of branch would address the ability of hardware to handle vector lane divergence, and software's ability to express divergent control flow. We expect that such experiments would be interesting in the context of AVX-512 and GPU programming models such as OpenCL and SYCL.

The PRKs have been the basis of a number of programming model evaluations in the past, including studies of Partitioned Global Address Space (PGAS) models [38], one-sided notification in both SHMEM [5] and MPI [4], Fortran coarray events [10, 11], multiple memory kinds in SHMEM [28], Chapel [19] and HPX [18]. Until recently, the primary focus of most users of the PRK was distributed-memory programming models [36] and subsequently an investigation of dynamic load-balancing features (or lack thereof) in popular distributed programming models [12, 35]. Thus, the current study reflects a significant departure from past work, in that it focuses exclusively on shared-memory and accelerator/offload programming models.

Figure 2: A pictoral representation of the PRK p2p kernel.



$$A_{i,j} = A_{i-1,j} + A_{i,j-1} - A_{i-1,j-1}$$

3 EXPERIMENTAL METHODOLOGY

The evolution of the PRK codes began with C89 implementations, which were subsequently cleaned up to use modern C style, including the removal of preprocessor macros that improved the readability of the code but which made it less adaptable to new syntax. The C++ implementations were based on sequential implementations that used std::vector for all array data structures. We subsequently found that the STL's inability to support NUMA via Linux's first-touch policy required us to use alternative allocation schemes to scale to multiple NUMA nodes. While there are solutions to this problem – e.g. Kokkos array [6, 7] – we do not consider it a primary concern, because most of applications exploit coarse-grain parallelism (e.g. MPI) that addresses NUMA more effectively than loop-level parallelism.

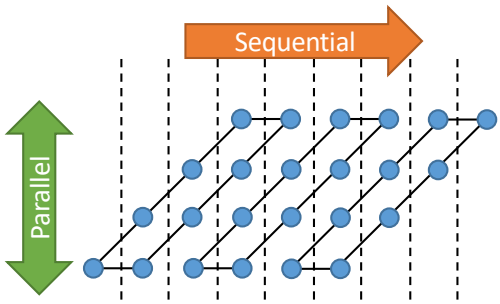
When porting the PRK codes to new programming models, we try to be idiomatic, i.e. write code that represents a reasonable usage of the programming model in question, rather than use a specific style of programming. At the same time, the PRK ports must be as similar as possible to one another to allow for objective performance comparisons. Fortunately, in the case of the C++ frameworks considered here, there is enough similarity that both goals could be met.

The easiest PRK to implement is nstream, as most of the implementations merely replace the sequential for loop with a parallel for template using a lambda for the loop body. For transpose, we use a two-dimensional parallel loop if one exists and nested parallel loops if one doesn't. Because transpose often benefits from loop blocking due to e.g. TLB misses, we also implemented a blocked version for every model. For example, we use Kokkos::MDRangePolicy, RAJA::KernelPolicy, tbb::parallel_for and tbb::blocked_range2d, and SYCL parallel_for with a 2D range. On the other hand, the sequential reference, the range-based for, and both the parallel and sequential versions using std::for_each use two or more nested loops.

To implement the stencil kernel, we employ a code generator to specialize for each case (radius and pattern). We expect that real applications are choosing the stencil pattern inside of the loop over all grid points, so this is an appropriate choice, although our use of constant coefficients may not be representative of all applications. Like transpose, all of the stencil implementations support a loop blocking parameter, although in this case the performance improvement associated therewith is due to data caches rather than TLBs.

The final kernel considered in this study is synch_p2p (henceforth called p2p). This kernel is not data-parallel in the same manner as the prior kernels, because the sequential version contains a loop-carried dependency that prevents the use of parallel loop constructs. However, p2p is data-parallel along the anti-diagonals of the matrix; this is how we realize parallelism in programming models that only support simple data parallelism. Unfortunately, parallelizing over the inner loop in the skewed version is not efficient, because it requires a barrier between every outer-loop iteration and because the inner loop must access data at very larger stride. To amortize synchronization overheads, we block the anti-diagonal loop, which requires $O(n/b)$ barriers, for a square matrix of dimension n and b-way blocking. Figure 3 has a graphical representation of the parallel version obtained from skewing the iteration space. Blocking

Figure 3: A pictoral representation of the PRK p2p kernel parallelized over the inner loop after skewing the iteration space. Synchronization can be amortized by doing barriers less frequently in the inner loop.



the inner (parallel) loop produces the hyperplane implementation. Another way to reduce synchronization is to block the loops of the original sequential version and enforce data dependencies at the block level. This requires that the programming model support tasks with dependencies. In our experiments, we implemented this only for OpenMP and TBB flowgraph. Figure 4 has a graphical representation of the implementation.

4 RESULTS

In this section, we evaluate the programming models under consideration both from a progammability perspective. At the end, we

Figure 4: A pictoral representation of the PRK p2p kernel where synchronization is amortized by tiling the loops and enforcing dependencies between tiles.

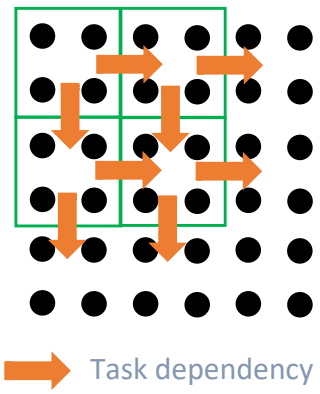


Figure 5: Data parallel syntax for some of the programming models considered. This source corresponds approximately to the nstream implementations, but many simplifications have been applied to fit into the available columns. All namespaces have been removed, but can be found explicitly in the GitHub version of the codes.

```
// Reference
for (int i=0; i<n; i++) BODY

// range-based for
// prk::range is either the Boost
// or Niebler implementation
auto r = prk::range(0,n);
for (auto i : r) BODY

// STL
auto r = prk::range(0,n);
for_each(par, begin(r), end(r), [&] (int i)
  BODY
);

// Kokkos
parallel_for(n, KOKKOS_LAMBDA(int i)
  BODY
);

// RAJA
forall<thread_exec>(0, n, [=](Index_type i)
  BODY
);

// TBB
blocked_range<int> r(0, n);
parallel_for(r, [&](decltype(r)&r) {
  for (auto i=r.begin(); i!=r.end(); ++i )
    BODY
);
```

will show that the performance of equivalent implementations is quite similar.

Figure 5 shows code samples extracted from the nstream kernel implementations. Not surprisingly, the syntax for a parallel loop is roughly the same, and models can be made more or less pairwise similar by changing idioms. For example, RAJA has a parallel_for that has an explicit range, which makes it look more similiar to our implementation using TBB. In the case of SYCL, we show only the parallel_for method, but this code cannot exist by itself.

Because SYCL exposes control over heterogeneous execution, the parallel_for method is scoped to the queue submit method, which ensures the compiler generates device object code. Figure 6 shows a more complete sample for SYCL.

Figure 6: A simplified representation of SYCL loop parallelism, as employed by the PRK nstream kernel. Here `d` and `h` are device (host) buffers, respectively, while `p` is the accessor used by the loop body to modify data.

```
queue q(cl::sycl::default_selector{});
buffer<T> d{ h.data(), h.size() };
q.submit([&](handler& h) {
  auto p = d.get_access<read_write>(h);
  h.parallel_for(range<1>{n}, [=] (item<1> i)
    BODY(p)
  );
});
q.wait();
```

In the course of implementing the PRK codes, some obvious differences emerged between the frameworks, which are summarized in Table 1. Obviously, all support simple data parallelism via a parallel for loop. Most frameworks support nested loops, although in the case of OpenCL and SYCL, only three nested loops are allowed (the programmer can obviously map more loops onto fewer, but this is tedious). The parallel STL doesn't support nested parallelism explicitly, but the `for_each` method nests properly, at least for the execution policies in use so far. Unfortunately, Boost.Compute does not make nested parallel easy, so we did not attempt to implement stencil or transpose with it. We were particularly discouraged because, while there is a transpose example in the Boost.Compute repository, it is merely an OpenCL program where the Boost.Compute interface is used to manage the OpenCL kernel invocation.

Neither OpenCL nor SYCL supports reduce or scan primitives, although these may be obtained from the Khronos parallel STL [13] or another library implementation of these. All of the other frameworks support reduce and scan, which is an indication of the importance of these primitives to HPC programmers. Finally, we note that OpenMP acquired scan support only in the latest version of the specification (5.0) [30].

Table 1: A high-level summary of the features present in the various C++ frameworks. See text for details.

Model	for	nested for	reduce	scan
TBB	Y	Y	Y	Y
C++17 PSTL	Y	N	Y	Y
RAJA	Y	Y	Y	Y
Kokkos	Y	Y	Y	Y
Boost.Compute	Y	N	Y	Y
SYCL	Y	3	N	N
OpenCL	Y	3	N	N
OpenMP	Y	Y	Y	Y

5 CONCLUSIONS

In this paper, we described the porting of the PRK codes to parallel C++ frameworks. The goal of this effort was to create a set

of interesting tests to evaluate the features and performance of programming models. Our results show that while all parallel models share the same basic feature of loop-level parallelism, there is a divergence between the Khronos models (SYCL and OpenCL) and other models (TBB, Kokkos, RAJA, OpenMP). This is easily understood from their different purposes – one can build reduce, scan and arbitrary nested loops on top of SYCL and OpenCL, and there are already open-source projects that do this (Khronos Parallel STL, which uses SYCL). On the other hand, RAJA, Kokkos, TBB and OpenMP are all used directly by HPC programmers who do not want another layer to get the features they use, because these frameworks are themselves layers on top of something else. For example, RAJA and Kokkos sit on top of OpenMP, CUDA, or other programming model required to support the hardware ecosystem associated with their user communities.

We did not examine performance in this paper because we are reporting on the development of a research tool that others can use for a range of activities, including the comparison of the performance of different implementations. The PRK experiments we have performed with the codes described in this paper have revealed a number of performance, performance-portability and functional issues with essentially all of the frameworks considered. For example, we found that TBB's task scheduler is more scalable on a processor that lacks a shared last-level cache, whereas OpenMP's centralized task scheduler is more efficient than TBB's when one is present. Initially, our Kokkos stencil code scaled better than all the others because Kokkos' containers handle NUMA properly, unlike the STL. This is not an STL implementation defect but specified behavior, so we have resolved to only measure thread scaling within a single NUMA node until such a time as we replace `std::vector` with a different data structure that behaves like Kokkos'. Initially, TBB stencil implementation was significantly faster than others prior to explicit cache-blocking, because TBB's `parallel_for` interface compels the user to write code that is naturally blocked. These are but a few of the interesting things that can be learned by porting the PRK codes and testing them on a range of platforms. We have only begun to scratch the surface of what can be learned on specialized hardware devices, although we have tested the GPU implementations of some of the programming models already.

ACKNOWLEDGMENTS

The authors cannot thank Rob van der Winjgaart enough for many years of fruitful collaboration on the Parallel Research Kernels; without his efforts, none of this would have been possible. We thank Pablo Reble for providing a high-quality implementation of the p2p kernel using the TBB flowgraph API. We thank Alex Duran for help understanding the OpenMP tasking API and to our colleagues that identified unnecessary synchronization in the OpenMP task implementation of p2p, as described in [3]. Xinmin Tian and Martyn Corden provided numerous suggestions on how to get the best results from the Intel compilers. The SYCL team at CodePlay, the TBB team at Intel, the RAJA team at Lawrence Livermore National Laboratory, and the Kokkos team at Sandia National Laboratory all provided helpful feedback as we were writing the respective implementations. We apologize to anyone else who contributed to the Parallel Research Kernel developments reported in this paper who is not mentioned. Finally, we thank Brad Chamberlain for thoughtful analysis of the PRK project and for unintentionally inspiring us to set about supporting new programming languages.

REFERENCES

[1] 2019. Parallel Research Kernels. https://github.com/ParRes/Kernels
[2] 2019. Travis CI – ParRes/Kernels. https://travis-ci.org/ParRes/Kernels
[3] Vishakha Agrawal, Michael J. Voss, Pablo Reble, Vasanth Tovinkere, Jeff Hammond, and Michael Klemm. 2018. Visualization of OpenMP* Task Dependencies Using Intel® Advisor – Flow Graph Analyzer. In *Evolving OpenMP for Evolving Architectures*, Bronis R. de Supinski, Pedro Valero-Lara, Xavier Martorell, Sergi Mateo Bellido, and Jesus Labarta (Eds.). Springer International Publishing, Cham, 175–188.
[4] R. Belli and T. Hoefler. 2015. Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 871–881. https://doi.org/10.1109/IPDPS.2015.30
[5] James Dinan, Clement Cole, Gabriele Jost, Stan Smith, Keith Underwood, and Robert W. Wisniewski. 2014. Reducing Synchronization Overhead Through Bundled Communication. In *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools*, Stephen Poole, Oscar Hernandez, and Pavel Shamis (Eds.). Springer International Publishing, Cham, 163–177.
[6] H. Carter Edwards and Daniel Sunderland. 2012. Kokkos Array Performance-portable Manycore Programming Model. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM '12)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/2141702.2141703
[7] H. Carter Edwards, Daniel Sunderland, Vicki Porter, Chris Amsler, and Sam Mish. 2012. Manycore Performance-portability: Kokkos Multidimensional Array Library. *Sci. Program.* 20, 2 (April 2012), 89–114. https://doi.org/10.1155/2012/917630
[8] H. C. Edwards and C. R. Trott. 2013. Kokkos: Enabling Performance Portability Across Manycore Architectures. In *2013 Extreme Scaling Workshop (xsw 2013)*. 18–24. https://doi.org/10.1109/XSW.2013.7
[9] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202 – 3216. https://doi.org/10.1016/j.jpdc.2014.07.003 Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.
[10] Alessandro Fanfarillo and Jeff Hammond. 2016. CAF Events Implementation Using MPI-3 Capabilities. In *Proceedings of the 23rd European MPI Users' Group Meeting (EuroMPI 2016)*. ACM, New York, NY, USA, 198–207. https://doi.org/10.1145/2966884.2966916
[11] Alessandro Fanfarillo and Davide Del Vento. 2017. Notified Access in Coarray Fortran. In *Proceedings of the 24th European MPI Users' Group Meeting (EuroMPI '17)*. ACM, New York, NY, USA, Article 12, 7 pages. https://doi.org/10.1145/3127024.3127026
[12] Evangelos Georganas, Rob F. Van der Wijngaart, and Timothy G. Mattson. 2016. Design and Implementation of a Parallel Research Kernel for Assessing Dynamic Load-Balancing Capabilities. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 73–82. https://doi.org/10.1109/IPDPS.2016.65
[13] The Khronos 'Group. [n. d.]. Open Source Parallel STL implementation. https://github.com/KhronosGroup/SyclParallelSTL
[14] Georg Hager. 2019. The McCalpin STREAM benchmark: How do do it right and interpret the results. https://blogs.fau.de/hager/archives/8263
[15] Richard D. Hornung and Jeffrey A. Keasler. 2014. The RAJA Portability Layer: Overview and Status. (9 2014). https://doi.org/10.2172/1169830
[16] Intel Corporation. [n. d.]. Threading Building Blocks (TBB). https://github.com/01org/tbb. https://www.threadingbuildingblocks.org/
[17] ISO. 2017. *ISO/IEC 14882:2017 Information technology — Programming languages — C++* (fifth ed.). International Organization for Standardization, Geneva, Switzerland. 1605 pages. https://www.iso.org/standard/68564.html
[18] Hartmut Kaiser, Thomas Heller, Daniel Bourgeois, and Dietmar Fey. 2015. Higher-level Parallelization for Local and Distributed Asynchronous Task-based Programming. In *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware (ESPM '15)*. ACM, New York, NY, USA, 29–37. https://doi.org/10.1145/2832241.2832244
[19] E. Kayraklioglu, W. Chang, and T. El-Ghazawi. 2017. Comparative Performance and Optimization of Chapel in Modern Manycore Architectures. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1105–1114. https://doi.org/10.1109/IPDPSW.2017.126
[20] Khronos OpenCL Working Group. 2012. The OpenCL Specification, Version 1.2, Aaftab Munshi (Ed.). https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf
[21] Lawrence Livermore National Laboratory. [n. d.]. RAJA Performance Portability Layer. https://github.com/LLNL/RAJA

[22] Sandia National Laboratory. [n. d.]. Kokkos C++ Performance Portability Programming EcoSystem: The Programming Model – Parallel Execution and Memory Abstraction. https://github.com/Kokkos/kokkos

[23] Piotr R. Luszczek, David H. Bailey, Jack J. Dongarra, Jeremy Kepner, Robert F. Lucas, Rolf Rabenseifner, and Daisuke Takahashi. 2006. The HPC Challenge (HPCC) Benchmark Suite. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*. ACM, New York, NY, USA, Article 213. https://doi.org/10.1145/1188455.1188677

[24] Devin Matthews. [n. d.]. TBLIS (Tensor BLIS). https://github.com/devinamatthews/tblis

[25] Timothy Mattson, Beverly Sanders, and Berna Massingill. 2004. *Patterns for Parallel Programming* (first ed.). Addison-Wesley Professional.

[26] John McCalpin. 1995. Memory bandwidth and machine balance in high performance computers. *IEEE Technical Committee on Computer Architecture Newsletter* (12 1995), 19–25.

[27] John D. McCalpin. 2015. STREAM: Sustainable Memory Bandwidth in High Performance Computers. https://www.cs.virginia.edu/stream/

[28] Naveen Namashivayam, Bob Cernohous, Krishna Kandalla, Dan Pou, Joseph Robichaux, James Dinan, and Mark Pagel. 2018. Symmetric Memory Partitions in OpenSHMEM: A Case Study with Intel KNL. In *OpenSHMEM and Related Technologies. Big Compute and Big Data Convergence*, Manjunath Gorentla Venkata, Neena Imam, and Swaroop Pophale (Eds.). Springer International Publishing, Cham, 3–18.

[29] OpenMP Architecture Review Board. 2015. OpenMP Aplication Program Interface – Version 4.5. https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[30] OpenMP Architecture Review Board. 2018. OpenMP Aplication Program Interface – Version 5.0. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[31] S. J. Plimpton, R. Brightwell, C. Vaughan, K. Underwood, and M. Davis. 2006. A Simple Synchronous Distributed-Memory Algorithm for the HPCC RandomAccess Benchmark. In *2006 IEEE International Conference on Cluster Computing*. 1–7. https://doi.org/10.1109/CLUSTR.2006.311859

[32] Gopalakrishnan Santhanaraman, Sundeep Narravula, Amith. R. Mamidala, and Dhabaleswar K. Panda. 2007. MPI-2 One-Sided Usage and Implementation for Read Modify Write Operations: A Case Study with HPCC. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Franck Cappello, Thomas Herault, and Jack Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 251–259.

[33] Tyler M. Smith, Robert A. van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee. 2014. Anatomy of High-Performance Many-Threaded Matrix Multiplication. In *28th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2014)*.

[34] Khronos® OpenCL™ Working Group SYCL™ subgroup. 2019. SYCL™ Specification. https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf, Ronan Keryell, Maria Rovatsou, and Lee Howes (Eds.).

[35] Rob F. Van der Wijngaart, Evangelos Georganas, Timothy G. Mattson, and Andrew Wissink. 2017. A New Parallel Research Kernel to Expand Research on Dynamic Load-Balancing Capabilities. In *High Performance Computing*, Julian M. Kunkel, Rio Yokota, Pavan Balaji, and David Keyes (Eds.). Springer International Publishing, Cham, 256–274.

[36] Rob F. Van der Wijngaart, Abdullah Kayi, Jeff R. Hammond, Gabriele Jost, Tom St. John, Srinivas Sridharan, Timothy G. Mattson, John Abercrombie, and Jacob Nelson. 2016. Comparing Runtime Systems with Exascale Ambitions Using the Parallel Research Kernels. In *High Performance Computing*, Julian M. Kunkel, Pavan Balaji, and Jack Dongarra (Eds.). Springer International Publishing, Cham, 321–339.

[37] Rob F. Van der Wijngaart and Timothy G. Mattson. 2014. The Parallel Research Kernels: A tool for architecture and programming system investigation. In *Proceedings of the IEEE High Performance Extreme Computing Conference*. IEEE. https://doi.org/10.1109/HPEC.2014.7040972

[38] Rob F. Van der Wijngaart, Srinivas Sridharan, Abdullah Kayi, Gabrielle Jost, Jeff R. Hammond, Timothy G. Mattson, and Jacob E. Nelson. 2015. Using the Parallel Research Kernels to Study PGAS Models. In *2015 9th International Conference on Partitioned Global Address Space Programming Models*. 76–81. https://doi.org/10.1109/PGAS.2015.24

[39] Field van Zee et al. [n. d.]. BLIS. https://github.com/flame/blis