The Parallel Research Kernels

A tool for architecture and programming system investigation

Rob F. Van der Wijngaart
Intel Labs
Intel Corp
Santa Clara, CA, USA
rob.f.van.der.wijngaart@intel.com

Timothy G. Mattson
Intel Labs
Intel Corp
Hillsboro, OR, USA
timothy.g.mattson@intel.com

***Abstract*: We present the Parallel Research Kernels; a collection of kernels supporting research on parallel computer systems. This set of kernels covers the most common patterns of communication, computation and synchronization encountered in parallel HPC applications. By focusing on these kernels instead of specific workloads, one can design an effective parallel computer system without needing to make predictions about the nature of future workloads.**

Keywords—parallel; kernel; high-performance computing; compact; verification

I. INTRODUCTION

To design a useful parallel computer you need workloads that represent the requirements of the intended users of that computer. The need for workloads has inspired a plethora of software test suites, ranging in complexity from simple kernels to full production-level applications. Test suites based on full applications capture the needs of current users, but as workloads evolve and user communities change, the relevance of these suites degrades. Our central thesis is that while nobody can predict with any certainty future workloads or even future users, we can be confident—based on the last 30 years of research in parallel computing—of the features of a parallel computer that will be stressed by future workloads. A test suite of programs focused on those features will remain relevant for decades. Furthermore, if these programs cover a wide enough range of such features, the test suite will be sufficient to support the design of future parallel systems.

We are especially interested in parallel, high-performance, scientific workloads, but also target certain non-scientific workloads. To guide the selection and implementation of these workloads we use the following design goals: *simple to compile and run* on any platform (including on simulators of new architectures), so no special library dependencies, and only plain C; *compact,* so they are easy to port to new platforms or new programming models; *no input files* (part of compactness, so any data set must be synthesized); *arbitrarily scalable* (problem size and number of processes or threads); *self-verifying* (a workload must do some useful work, whose results are sensitive to any errors in implementation, tools, or runtime); *generic implementation* not tuned to a particular architecture; allowing formulation of a *simple performance model* (`performance expectation'); *uniform* in coding style and coverage (serial, OpenMP, and MPI versions of the whole suite); *load balanced* (an unbalanced load does not provide insight into the architecture).

We believe we have achieved these design goals; the results are archived as an Open Source project [1] under the name of the Parallel Research Kernels (PRK)

II. RELATED WORK

A number of efforts have been made in the past to create comprehensive suites of test codes or benchmark programs. We point specifically to the HPC Challenge suite [2], which contains a number of well-known benchmarks. This suite is also arbitrarily scalable, but does not have the breadth or the compactness, of PRK. We adapted some of the HPCC test codes for inclusion in the PRK. Another well-known suite is the NAS Parallel Benchmarks [3]. Its self-verifying property has proved very useful to debug new tools and compilers, but due to the complexity of the components of the suite, such verification has only been possible for a discrete set of problem sizes. The CLOMP benchmark [4] measures OpenMP overheads within a workload that can be scaled in various dimensions. It gives much insight into such overheads within a particular application area, but does not elucidate potential performance obstacles in other areas (nor does it claim to do so). It also does not feature strict correctness checking. The EPCC suite [5] also measures OpenMP overheads, but in a more synthetic environment than CLOMP and PRK, and counts as strictly micro-benchmarks, without verification or correspondence to a realistic workload scenario. Both HPCC and NPB have serial, OpenMP, and MPI implementations, like PRK. But since their codes are on the order of 10 times larger than PRK codes, they are much harder to port to new programming models. Other much-used HPC suites are SPEC MPI2007 and SPEC OMP2012 [6]. These concern problems of a single size, and there is no overlap between the MPI and OpenMP suites, making comparisons difficult.

The rest of this paper is organized as follows. In Section III we describe some of the unique features of PRK, including the simple performance models we derived for the kernels. In Section IV we introduce all the kernels at a high level. Section V describes some of the kernels in more detail. In Section VI we present some significant use cases of PRK. We present a summary and suggestion for future work in Section VII.

978-1-4799-6233-4/14/$31.00 ©2014 IEEE

III. UNIQUE PRK FEATURES

A unique feature of PRK is the inclusion of performance expectations, which are very simple performance models, based on just a few system and problem parameters. They gloss over many details, but serve as sanity checks on the system performance. While the implementations of the kernels are all completely the analytical formulas manageable. Recognizing architectural differences, we provide expectations for clusters as well as for many-core chips, where the latter are parameterized and thus allow us to investigate different stress points of the computer system, the performance expectations are only provided for asymptotic cases of very small and very large problems, which keeps assumed to provide hardware-supported cache coherence. Future extensions will include combinations of the two, i.e. clusters of many-core chips.

We use the following definitions in the expression of our performance expectations. We use the symbols α and β for latency and bandwidth, respectively. Aggregate (bi-section) bandwidth is $\underline{\beta}$. Subscripts "M" and "$" refer to memory and cache (line), respectively. Subscripts "N" and "C" refer to (cluster) Node(s) and Chip (or processor), respectively. "L" stands for Length.

Table 1: Symbols for performance expectations

Symbol	Meaning
β_M	memory bandwidth (one core/node to memory)
α_M	memory latency
α_N	point-to-point message latency
β_N	point-to-point message bandwidth
$\underline{\beta}_N$	aggregate (bi-section) message bandwidth
$\underline{\beta}_{CM}$	aggregate memory bandwidth of processor
L_W	word size
$L_\$$	cache line length
P	number of cores/cluster nodes
FP	peak scalar/vector FLOPs (floating point multiply-and-add/s) per core/node
IP	peak scalar/vector integer performance (instructions/s) per node/core
$T_{S,L}$	execution time for small or large problem size

We note that β_N may be of the same order as β_M, depending on the system configuration and implementation. However, α_N will normally be much larger than α_M, even in the case of shared memory systems, due to software overheads. We assume that on-chip cache-to-cache transfers are much faster than accesses to off-chip memory.

We assume that the processor cores feature fused multiply-add floating point pipelines, so that maximum FP speed is only reached for computations that have a balanced mix of multiplications and additions.

We ignore latency and bandwidth between the various levels of cache and registers within a specific node or core, these are considered irrelevant compared to traffic between memory and cache.

Another special feature of PRK is a kernel that explicitly tests efficiency of certain operations/constructs that are important for a great number of realistic scientific workloads, parallel or not, and that do not receive attention in other suites. This kernel is called 'branch,' and will be described in Section V.C.

PRK tries to balance simplicity and compactness of operations with the need always to do some useful work, which can be verified for correctness. That is, the specification is written in terms of the work that needs to be done, not the control or communication structures to achieve it. For example, some OpenMP micro-benchmarks may measure the time it takes to execute a sequence of barriers, but these would never occur in a real application without intervening modifications of values shared among threads. In addition, it would not be verifiable whether or not one or more of the barriers might have been skipped.

Most of the PRK are formulated such that the main operation is carried out in an iterative loop. We take special care to make sure that there is a dependence from one iteration to the next, so that the verification test fails if an iteration is skipped, or if the result of an earlier iteration is incorrect. The exception is the branch kernel (see Section V.C).

IV. THE KERNELS

In this section, we list the kernels included in the PRK and provide a high-level description of the system features stressed by each kernel.

Table 2: Summary of kernels

Name	Function
Transpose	Dense matrix transpose
Reduce	Element-wise sum of multiple vectors
Sparse	Sparse matrix-vector product
Random	Random updates to table
Synch_global	Global synchronization
Synch_p2p	Point-to-point synchronization
Stencil	Stencil operation
Refcount	Updating shared or private counters
Nstream	Daxpy
Dgemm	Dense matrix-matrix product
Branch	Inner loop with branches

Transpose stresses communication and memory bandwidth. It features unit stride on read and regular but non-unit stride on write (or the other way around).

Reduce also stresses communication and memory bandwidth, as well as synchronization efficiency.

Sparse is dominated by indirect memory references with unknown stride on read (see Section V.B)

Random (derived from the Random Access kernel in HPCC [2]) experiences irregular memory access on both read and write as elements in a large table are updated pseudo-randomly.

Synch_global concerns a somewhat unusual problem. Given a character string of a certain length, each thread/process selects a subset of its characters and compacts them. Next, all threads/processes collaborate to concatenate their substrings. This requires rather different collectives in OpenMP than in MPI to achieve the same functional result. The operation stresses the communication network (on- or off-chip).

Synch_p2p implements a stencil code with a demanding data dependence that is typically resolved using a fine-grain software pipeline technique. This operation is communication latency bound.

Stencil applies a data-parallel stencil operation to a two-dimensional array. It features multiple streams of regular but different strides on read, which benefits from efficient prefetching.

Refcount demands efficient mutual exclusion (see Section V.A).

Nstream, derived from Stream [7], is embarrassingly parallel; its performance is dominated by memory bandwidth.

Dgemm, if implemented carefully, should only be limited by peak floating point performance for large enough matrices.

Branch features conditional and unconditional inner-loop branches that stress, among others, compiler technology, SIMD vectorization quality, instruction cache size and efficiency, and linker efficiency. It is embarrassingly parallel (see Section V.C)

V. DETAILED EXAMPLES

We describe three of the PRK in more detail to illustrate our general approach, and to highlight some of the unique PRK features.

A. The *Refcount* Kernel

Usage of shared read/write variables is specific to the shared-memory programming model. Traditionally, locks and critical sections are used when groups of statements or statements with side effects must be executed atomically to protect against data races. Often they are used in the process of parallelizing a code to guarantee correctness. Locks may severely degrade performance, but if used judiciously, they can help speed up the programming process without undue application slowdown. In properly constructed applications with significant use of locks, many lock accesses will be uncontended. Optimizing for this situation, however, may lead to suboptimal treatment of the contended lock, causing potentially disproportionate slowdown of applications in which the locks are active. Hence, both contended and uncontended locks must be evaluated.

An alternative to locks is offered by transactional memory, which supports grouped atomic operations with light-weight synchronization and rollback.

1) Shared reference counters

Work: All threads update a pair of shared counters in tandem N times; this prohibits the use of the atomic directive in OpenMP, which only protects updates of a single memory location. The counters may not be stored in adjacent memory locations, which would allow a hardware-assisted atomic update of a single large word. In our reference code we store the members of the pair on distinct memory pages. Performance of this kernel is governed by the support for atomic transactions.

Performance expectation: Assumptions:

- If the multi-core system supports atomic read-modify-write semantics, acquiring and releasing a shared lock each requires one memory latency, using, for example MCS-style list-based queueing locks . If not, then Lamport's Bakery Algorithm [8] shows that 5 memory latencies are sufficient.
- The lock is fair, which means that after the threads queue their first request for the lock guarding the counters, they will be served in order, and after each lock acquisition and release the thread will enter the queue at the end. This means that a different thread will update the counters upon each lock acquisition.
- Updating the two reference counters can be overlapped with memory operations, but the counters themselves must be read from memory before we can update them.

Expected time (multi-core): $T_L = 4*\alpha_M*N*P$

2) Private reference counters

Work: All threads update a pair of private counters, which are stored in a shared array, whose rows are indexed by thread number. While any locks can be removed without affecting correctness, the compiler will not do so because the array is shared. Performance of this kernel is governed by the effectiveness with which the system supports non-conflicting atomic transactions.

Performance expectation: Assumptions:

- Reference counters as well as lock variables are kept in registers.
- Acquiring and releasing the private lock variables and updating the reference are integer operations. Each counts as a single, non-vectorizable integer operation.

Expected time (multi-core): $T_L = 4*N/IP$

Note that these two variations of the Refcount kernel do not specify that locks must be used—though our OpenMP reference implementation does. In case of the shared counters, mutual exclusion is truly necessary for correct execution, but we do not specify how this is accomplished.

B. The *Sparse* Kernel

Sparse matrix-vector products occur in many scientific applications. Optimized versions are frequently provided in mathematical libraries.

This kernel generates a, non-symmetric matrix of a given sparsity (fraction of non-zero matrix elements), and a dense vector. Then it multiplies a dense vector by the sparse matrix in parallel. Depending on the size of the vector and the sparsity, performance will be governed by inter-core bandwidth and memory latency. The main performance obstacle is the fact that vector elements are accessed in a more or less random order, reducing locality and making hardware prefetching hard.

The sparse matrix is built as follows. The standard star-shaped difference stencil with a user-specified radius is applied to a scalar variable on a 2-dimensional, square grid. Example of a stencil operation with radius 2 (r=2):

$a(i,j) = c_1*b(i-2,j) + c_2*b(i-1,j) + c_3*b(i,j) + c_4*b(i+1,j) + c_5*b(i+2,j) + c_6*b(i,j-2) + c_7*b(i,j-1) + c_8*b(i,j+1) + c_9*b(i,j+2)$

Here the coefficients c_1 through c_9 are constants. Non-overlapping arrays a and b signify field variables defined on the grid, indexed by grid point indices (i,j). If we linearize the index space of a and b in the canonical fashion, we can write the stencil operation as: a = M b, where M is the sparse matrix of interest.

A square grid with linear dimension 2^n has $2^{2n} = 4^n$ points. Hence, M has 4^n rows and 4^n columns, for a total of 16^n elements (most elements are zero; only nonzero elements are stored). The stencil is applied in a periodic fashion, i.e. it wraps around the edges of the grid. The columns of M are permuted in a pseudo-random way, based not on a random seed that is different for each thread/process, but on bit swapping of array indices. The result is a general, irregular sparse matrix, but with a known number of 4r+1 nonzeroes per row. Generating the sparse matrix in the described way has multiple advantages:

- The statistical properties of the matrix are constant as the size grows.
- Unlike in the case of general sparse matrices, the load can be balanced perfectly, due to the fact that each row has the same number of nonzeroes.
- Generated the matrix in parallel is trivial, and produces the same result, regardless of the number of participating threads/processes.

We use Compressed Row Storage (CRS) for accessing the matrix elements. Numerical values of matrix and vector elements are chosen judiciously to make verification easy. They do not correspond to a physical discretization of a continuum problem.

Performance expectation: Assumptions:

- Requests for the irregular reads of vector elements can be pipelined, so we can ignore memory latency, but only one word out of each cache line is used.
- Long vectors: because there is negligible reuse, performance is dominated by memory traffic, and computational cost can be ignored, as long as the FPU units are properly pipelined.
- CRS requires the storage and reading of indices to be used for indirect referencing. We take into account the bandwidth requirement of reading these indices, but assume that an index is as long as a floating point word, for convenience.
- Short vectors: All data is assumed to be already present in cache, and we ignore the cost of loading and storing data for both the vector initialization part and the matrix-vector multiplication part.
- Cluster: the method uses static domain decomposition (by rows) of the matrix and replicates the vector, so no communications are involved in doing the actual multiplication, but each process initializes or increments its own chunk of the vector and broadcasts that to the other processes.
- Multi-core: each thread initializes or increments its own chunk of the vector before each multiplication.

MULTI-CORE:

Long vectors: For each of the 4^n rows of the matrix a multiplication involves reading 4r+1 words randomly from the vector, requiring (4r+1) cache line loads. It also requires reading 4r+1 indices (stride one), 4r+1 matrix elements (stride one), one read of the result vector element (stride one), and one write of the result vector (stride 1). For the vector initialization we incur one write (stride one) per row of the matrix. Since all cores will be loading and storing simultaneously, we use the aggregate memory bandwidth to evaluate the cost of message traffic.

$$T_L = 4^n[(4r+1)(L_\$+2L_W)+3L_W]/\underline{\beta}_{CM}$$

CLUSTER:

Long vectors: Each process owns $4^n/P$ rows of the matrix and the same number of vector elements. Hence, we can compute the non-communication part of the execution time based on the multi-core evaluation. The communication involves broadcasting the locally generated vector segments to all processes. For long vectors the optimal algorithm communicates (P-1) complete vectors in the aggregate in a number of stages (see Reduction kernel). We use the cluster bi-section bandwidth to compute the cost of that data traffic.

$$T_L = 4^n/P*[(4r+1)(L_\$+2L_W)+3L_W]/\beta_M + 4^n(P-1)*L_W/\underline{\beta}_N$$

C. The *Branch* Kernel

While synthetic benchmarks may feature regular or irregular data access, they often are quite regular and predictable in terms of code paths, especially inside loops or loop nests that contain most of the work. Real applications, however, often feature inner loop data-dependent branches. These can defeat speculative execution and prefetching, which might otherwise return sizeable performance improvements. Branch-intensive loops can be divided into several broad categories: a) executed statements differ, depending on the branch(es) taken, but the memory references do not change; b) executed statements differ, depending on the branch taken, as

well as the memory locations referenced; c) each branch choice causes a call to a different substantial function, causing potentially large numbers of instructions to be loaded into the instruction cache and discarded from it at high frequency; d) each branch choice corresponds to an alternative, with no or merely inlined calls to short functions. These categories are not all mutually exclusive.

In this kernel we focus on four examples, drawn from the above categories. The first three all concern light-weight loops (of length L), i.e. loops whose iterations have very few instructions associated with them (category d).

1. Branches inside vectorizable loops, where the introduction of the branch does not necessarily inhibit SIMD vectorization.
2. Branches inside vectorizable loops where the introduction of the branch does inhibit SIMD vectorization.
3. Branches inside non-vectorizable (SIMD) loops (even without the branch).
4. Branches inside loops in which each branch corresponds to a sizeable unique set of instructions.

Approach: For the light-weight loops we select the loop bodies as follows, where i is the loop index, expr1 is a very simple arithmetic expression, expr2 is an expression that selects a single element from an array, and expr3 is functionally the identity. All arrays and constants are of the integer data type.

```
aux = expr1(i);
if (expr2(i)>0)    vector[i] -= 2*vector[expr3(i)];
else                          vector[i] -= 2*aux;
```

Expr1(i) is the same for all three cases, and numerically equals vector[i] upon entry of the loop. Expr2(i) equals either aux (cases 1 and 3), or vector[index[i]] ≡ vector[i] (case 2). Expr3 in the first assignment of the loop body equals i (cases 1 and 2), or index[i] ≡ i (case 3). Array "vector" has approximately the same number of positive and negative elements. Sign changes occur approximately every 3 to four loop iterations. Each version of the loop with a conditional branch has a counterpart that does not contain the branch, but which does the same amount of computational work. Specifically, the loop body then gets replaced with:

```
aux = expr1(i);
vector[i] -= 2*(vector[expr3(i)]+aux);
```

The result of these choices is that the three different tasks all do exactly the same amount of computational work, so that the impact of the different types of branches can be compared. Moreover, the branches are "unpredictable," meaning that if the compiler guesses them to be always taken or to be always not taken, it will be wrong about 50% of the time. This ensures that the cost of branch misspredictionwill be measurable.

Example 4 is drawn from category c. Expr1(i) is now not a simple expression, but is obtained as follows. A square matrix A of user-defined size is filled element by element, such that the resulting matrix is numerically always the identity, but the actual instructions to compute the elements are different for different values of i (modulo a prescribed value). Subsequently, the matrix B is computed as the arithmetic average of A and A^T. The returned value of expr1 is i if B equals the identity matrix, and zero otherwise. Finally, we compute the vector element in the same way as the non-branching version of cases 1 and 2.The non-branching version of the loop is essentially the same as the branching version, except that the construction of matrix A is now no longer dependent on the value of i.

The result of these choices is that an amount of program text proportional to N*N, where N is the matrix order chosen, is generated for each different branch in the loop.

Performance expectation: Assumptions:

- Integer operations are not pipelined, they complete in the integer units in 1 cycle.
- A comparison counts for one integer operation.
- Indexing into an array (address computation) counts for one integer operation. That does not include the computation of the index itself.
- Each iteration of 1), 2), 3) does 7.5, 5.5, and 7 integer operations, respectively, on average (depends on outcome of test). 2) vectorizes (with masking), but 1) and 3) do not. Masking means that 2) will execute more instructions (7.5) than are actually used, but they are vectorized.
- The data set size for 1), 2) and 3) is constant and small, and the number of instructions involved is negligible, even for large problems (many invocations of the loop). Hence, for these problems instructions and data are all cached, so we do not need to take into account the cost of memory access.
- Each iteration of 4) does $(24+2*N^2)$ integer operations
- Each function of 4) contains K*N*N bytes of instructions (K depends, among others, on the level of optimization), which have to be read from memory for large problems. We assume that the reading of these instructions can be overlapped with computations.

MULTI-CORE:

1. $T_S = T_L = 7.5*L/IP$
2. $T_S = T_L = 7.5*L/IP$
3. $T_S = T_L = 7*L/IP$
4. $T_S = L*(24+2*N^2)/IP$
   $T_L = L*\max((24+2*N^2)/IP, K*N^2*P/\beta_{CM})$

CLUSTER:

1), 2), 3) and 4) (T_S) are identical to the multi-core expectations

4. $T_L = L*\max((24+2*N^2)/IP, K*N^2/\beta_M)$

VI. PRK USE CASES

Because of their portability, compactness, breadth, and arbitrary scalability, the PRK have been used in many projects inside Intel.

One project is the development of an experimental processor architecture that Intel has carried out for the US Department of Energy [9], for which various programming models were developed, implemented, and tested with the PRK. In addition, we have used PRK for validation of experimental compilers and runtimes.

Currently we are using the PRK for assessing expressivity and efficiency of two related task-based programming models, Concurrent Collections [10] and the Open Community Runtime [11].

In addition, the PRK are being considered for production environments like Intel Cluster Ready [12] for system-level evaluation and analysis.

Finally, we mention a recent study [13] that used the PRK to demonstrate the impact of a proposed extension to the OpenSHMEM programming interface. OpenSHMEM is the standard specification and library API to support one-sided communication, e.g. put and get operations, in a partitioned global access space (PGAS). In OpenSHMEM synchronization can be achieved using a collective barrier, or by polling or waiting on a flag that will be remotely updated using a one-sided operation. The proposed extension consists of *counting puts,* which effectively enable receiver-side synchronization as opposed to the current sender side synchronization. This greatly reduces synchronization overhead and latency costs. For the purpose of their study the authors ported the Synch_p2p PRK to OpenSHMEM. Synch_p2p implements a one-dimensional, fine-grain software pipeline that stresses point-to-point latency. It is implemented using 2-sided MPI communication based in MPI_Isend/MPI_Irecv. The first step of porting to it to OpenSHMEM was to replace 2-sided MPI communication by 1-sided MPI communication based on MPI_Put/MPI_Get. This was then easily ported to OpenSHMEM employing shmem_putmem.The authors found that the MPI implementation was an excellent starting point for the porting, which was accomplished in a matter of hours.

The ported Synch_p2p PRK kernel was used to compare existing *put* and *counting put* execution. Experiments were conducted on a 15-node cluster with a Mellanox® QDR InfiniBand interconnect. Each node in this cluster is configured with 24GB of memory and two Intel® Xeon™ X5680 processors, for a total of 12 cores per node, each supporting two hyperthreads, for a total of 24 hardware threads per node. Only four cores per node were used in this experiment. GFLOPS as a function of the number of cores (processing elements, PEs) is reported in figure VIII-1 (used with permission from the authors [13]).

VII. Summary and Future Work

The Parallel Research Kernels are a new, comprehensive suite of compact test programs that have been and are being used to assess various dimensions of merit of parallel computing systems. Due to their compactness and minimal requirements, they are extremely portable and have been used in a great variety of environments for a multitude of purposes. They mostly reflect the programming constructs found in high-performance scientific computing workloads. To follow up on the work described here, we will validate our performance expectations on multiple platforms, most notably on large-scale clusters and on many-core processors. In addition, we plan to extend the PRK to reflect constructs typical for Big Data workloads.

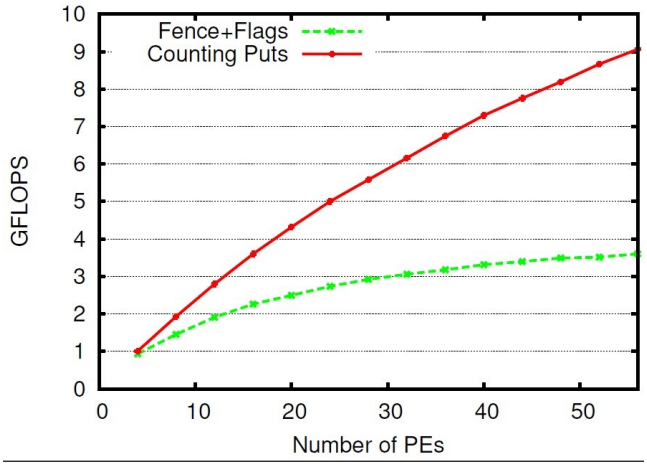

Figure VI-1: Synch_p2p PRK kernel ported to OpenSHMEM and running on a 15 node InfiniBand Cluster [13].

VIII. References

[1] https://github.com/ParRes/Kernels

[2] P. Luszczek, J.. Dongarra, D. Koester, R. Rabenseifner, R. Lucas, J. Kepner, J. McCalpin, D. Bailey, D. Takahashi, Introduction to the HPC Challenge Benchmark Suite, SC '06 Proceedings of the 2006 ACM/IEEE conference on Supercomputing

[3] D.Bailey, E. Barszcz, J.Barton, D.Browning, R.Carter, L. Dagum, R.Fatoohi, P. Frederickson, T.Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, S.Weeratunga, The NAS parallel benchmarks, International Journal of High Performance Computing Applications, 1991

[4] G. Bronevetsky, J. Gyllenhaal, B. de Supinski, CLOMP: Accurately Characterizing OpenMP Application Overheads, Int J Parallel Prog (2009) 37:250–265

[5] J. M. Bull, D. O'Neill, A microbenchmark suite for OpenMP 2.0, SIGARCH Comput. Archit. News, vol. 29, no. 5, pp. 41–48, 2001.

[6] http://www.spec.org/benchmarks.html

[7] J. McCalpin. STREAM: Measuring sustainable memory bandwidth in high performance computers, www.cs.virginia.edu/stream, 1995.

[8] L. Lamport, A New Solution of Dijkstra's Concurrent Programming Problem, Communications of the ACM 17, 8 (August 1974), pp. 453-455

[9] https://asc.llnl.gov/fastforward/

[10] A. Chandramowlishwarany, K. Knobe, R. Vuduc, Performance Evaluation of Concurrent Collections on High-Performance Multicore Computing Systems, IEEE International Symposium on Parallel & Distributed Processing (IPDPS), 2010

[11] https://01.org/open-community-runtime

[12] https://software.intel.com/en-us/cluster-ready

[13] J. Dinan, C. Cole, G. Jost, S. Smith, K. Underwood, R.Wisniewski, Reducing Synchronization Overhead Through Bundled Communication, Lecture Notes in Computer Science, Springer Verlag, Volume 8356, pp. 163-177, 2014