Investigating the Performance and Productivity of DASH Using the Cowichan Problems

Karl Fürlinger, Roger Kowalewski, Tobias Fuchs and Benedikt Lehmann
Ludwig-Maximilians-Universität (LMU) Munich
Computer Science Department, MNM Team
Oettingenstr. 67, 80538 Munich, Germany
Email: first.last@nm.ifi.lmu.de

ABSTRACT

DASH is a new realization of the PGAS (Partitioned Global Address Space) programming model in the form of a C++ template library. Instead of using a custom compiler, DASH provides expressive programming constructs using C++ abstraction mechanisms and offers distributed data structures and parallel algorithms that follow the concepts employed by the C++ standard template library (STL).

In this paper we evaluate the performance and productivity of DASH by comparing our implementation of a set of benchmark programs with those developed by expert programmers in Intel Cilk, Intel TBB (Threading Building Blocks), Go and Chapel. We perform a comparison on shared memory multiprocessor systems ranging from moderately parallel multicore systems to a 64-core manycore system. We additionally perform a scalability study on a distributed memory system on up to 20 nodes (800 cores). Our results demonstrate that DASH offers productivity that is comparable with the best established programming systems for shared memory and also achieves comparable or better performance. Our results on multi-node systems show that DASH scales well and achieves excellent performance.

ACM Reference Format:
Karl Fürlinger, Roger Kowalewski, Tobias Fuchs and Benedikt Lehmann. 2018. Investigating the Performance and Productivity of DASH Using the Cowichan Problems. In *HPC Asia 2018 WS: Workshops of HPC Asia 2018, January 31, 2018, Chiyoda, Tokyo, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3176364.3176366

1 INTRODUCTION

As computer systems are getting increasingly complex, it is becoming harder and harder to achieve a significant fraction of peak performance. This is true at the very high end, where supercomputers today are composed of hundreds of thousand compute cores and incorporate CPUs and GPUs in heterogeneous designs, but also at the low end, where application developers have to deal with shared memory systems of increasing scale (number of cores) and complex multi-level memory systems. Top-of-the-line server CPUs will soon have up to 32 physical cores on a single chip and many-core CPUs such as Intel's Knights Landing Xeon Phi architecture consists of up to 72 cores.

Of course, performance is only one side of the story and for many developers productivity is at least as big a concern as performance. This is especially true in newer application areas of high performance and parallel computing such as the life sciences or the digital humanities, where developers are less willing to spend large amounts of time to write and fine-tune their parallel applications.

In this paper we evaluate the performance and productivity characteristics of DASH, a realization of the PGAS paradigm in the form of a C++ template library. We start with an evaluation on shared memory systems, where we compare DASH to results obtained using Intel Cilk, Intel TBB, Go, and Chapel. To conduct our comparisons we rely on ideomatic implementations of the Cowichan problems by expert programmers that were developed in the course of a related study by Nanz et al. [17]. In addition, we perform a scalability study on a distributed memory system on up to 20 nodes (800 cores). Our results demonstrate that DASH offers productivity that is comparable with the best established programming systems for shared memory and also achieves comparable or better performance.

The rest of this paper is organized as follows: In Sect. 2 we provide background on the material covered in this paper by providing a short general overview of DASH and the Cowichan problems. In Sect. 3 we then describe the implementation of the Cowichan problems in DASH and compare the programming constructs employed with those used by expert programmers for their Go, Chapel, Cilk and TBB implementation. In Sect. 4 we provide an evaluation of the performance and productivity of DASH compared to these other implementations on a shared memory system and we provide a scaling study of the DASH code on multiple nodes of a supercomputing system. Sect. 5 discusses related work and in Sect. 6 we conclude and provide an outlook on future work.

HPC Asia 2018 WS, January 31, 2018, Chiyoda, Tokyo, Japan
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6347-1/18/01...$15.00
https://doi.org/10.1145/3176364.3176366

2 BACKGROUND

This section provides background information on our PGAS programming system DASH and the set of benchmark kernels we have used to evaluate its productivity and performance.

2.1 DASH

DASH [8] is a C++ template library that offers distributed data structures and parallel algorithms. A basic data structure available in DASH is the distributed array (`dash::Array`) that can span the memory of multiple processes (called execution *units* in DASH terminology), potentially running on separate interconnected nodes. Each unit can access all elements of the distributed array. If remote data is accessed, one-sided operations are triggered using the DASH runtime system (DART) [25]. DART is implemented on top of MPI-3 RMA (remote memory access) [15] using passive target synchronization mode. One-sided access operations are implemented using `MPI_Put` (writing a remote value) or `MPI_Get` (reading a remote value), while the remote (target) unit is never involved in the data transfer operation. This mode of one-sided access maps well to remote direct memory access (RDMA) technology, which is supported by every modern interconnect network and is even used for the implementation of the classic two-sided (send-receive) model in MPI [9].

Access to local data elements can be performed using direct memory read and write operations and the local part of a data structure is explicitly exposed to the programmer using a *local view* proxy object. For example, `arr.local` represents all array elements stored locally and access to these elements is of course much faster than remote access. Whenever possible, the *owner computes* model, where each unit operates on its local part of a data structure, should be used for maximum performance in DASH.

Besides one-dimensional arrays, DASH also offers shared scalars (`dash::Shared`) and multidimensional arrays (`dash::NArray`). Other data structures, notably dynamic (growing, shrinking) containers such as hash maps, are currently under development. DASH offers a flexible way to specify the data distribution and layout for each data structure using so-called data distribution patterns (`dash::Pattern`). For one-dimensional containers the data distribution patters that can be specified are cyclic, block-cyclic, and blocked. In multiple dimensions, these specifiers can be combined for each dimension and in addition tiled distributions are supported.

DASH also supports the notion of *teams*, i.e., groups of units that form the basis of memory allocations and collective communication operations. New teams are built by splitting an existing team starting with `dash::Team::All()`, the team that contains all units that comprise the program. If no team is specified when instantiating a new data structure, the default team `dash::Team::All()` is used.

DASH generalizes concepts found in the C++ Standard Template Library (STL). STL offers containers and algorithms, which are coupled using an iterator interface. Similarly, DASH data structures offer global iterators (`arr.begin()`, `arr.end()`) and local iterators (`arr.local.begin()`, `arr.local.end()`).

```cpp
#include <iostream>
#include <libdash.h>

int main(int argc, char *argv[]) {
  dash::init(&argc, &argv);

  // 2D integer matrix with 10 rows, 8 cols
  // default distribution is blocked by rows
  dash::NArray<int, 2> mat(10, 8);

  for (int i=0; i<mat.local.extent(0); ++i) {
    for (int j=0; j<mat.local.extent(1); ++j) {
      mat.local(i, j) = 10*dash::myid()+i+j;
    }
  }

  dash::barrier();

  auto max = dash::max_element(mat.begin(), mat.end());

  if (dash::myid() == 0) {
    print2d(mat);
    cout << "Max is " << (int)(*max) << endl;
  }

  dash::finalize();
}
```

Compile and Run:

```
$> mpicc -L ... -ldash -o example example.
    cc
$> mpirun -n 4 ./example
```

Output:

```
 0  1  2  3  4  5  6  7
 1  2  3  4  5  6  7  8
 2  3  4  5  6  7  8  9
10 11 12 13 14 15 16 17
11 12 13 14 15 16 17 18
12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27
21 22 23 24 25 26 27 28
22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37

Max is 37
```

Figure 1: A basic example DASH program (left) and its output when run with four execution units (right).

Thus STL algorithms (e.g., `std::fill`, `std::iota`) can be used in conjunction with DASH containers. DASH additionally provides parallel versions of selected algorithms. For example, `dash::fill` takes a global range (delimited by global iterators) and performs the fill operation in parallel.

DASH also generalizes the concepts of pointer, reference, and memory allocation found in regular C++ programs by offering global pointers (`dash::GlobPtr`), global references (`dash::GlobRef`), and global memory regions (`dash::GlobMem`). These constructs allow DASH to offer a fully-fledged PGAS programming system akin to UPC (Unified Parallel C) [7] while not requiring a custom compiler.

Fig. 1 shows a basic DASH program using a 2D array (matrix) data structure. The data type (int) and dimension (2) are compile-time template parameters, the extents in each dimension are set at runtime. In the example a (10×8) matrix is allocated and distributed over all units (since no team is specified explicitly). No specific data distribution pattern requested, so the default distribution by block of rows over all units is used. When run with four units, each unit gets $\lceil 10/4 \rceil = 3$ matrix rows, except for the last unit, which receives only one row.

Lines 11 to 15 in Fig. 1 show data access using the local matrix view by using the proxy object `mat.local`. All accesses are performed using local indices (i.e., `mat.local(1,2)` refers to the element stored locally at position `(1,2)`) and no communication operation is performed. The barrier in line 17 ensures that all units have initialized their local part of the data structure before the `max_element()` algorithm is used to find the maximum value of the whole matrix. This is done by specifying the global range that encompasses all matrix element (`mat.begin()` to `mat.end()`). In the library implementation of `max_element()`, each unit determines the locally stored part of the global range and performs the search for the maximum there. Afterwards a reduction operation is performed to find the global maximum. The return value of `max_element()` is a global reference for the location of the global maximum. In lines 21 to 24, unit 0 first prints the whole matrix (the code for `print2d()` is not shown) and then outputs the maximum by dereferencing the global reference `max`.

The right part of Fig. 1 shows the output produced by this application (bottom) and how to compile and run the program (top). Since DASH is implemented on top of MPI, the usual platform-specific mechanisms for compiling and running MPI programs are used. The output shown is from a run with four units (MPI processes), hence the first set of three rows are initialized to $0 \dots 9$, the second set of three rows to $10 \dots 19$, and so on.

2.2 The Cowichan Problems

The Cowichan problems [23] (named after a tribal area in the Canadian Northwest) are a set of small benchmark kernels that have been developed primarily for the purpose of assessing the usability of parallel programming systems. There are two versions of the Cowichan problems and we restrict ourselves to a subset of the problems found in the second set. Our work is based on previous work by Nanz et al. [17] in that we use the code publicly available from their study [1] to compare with our implementation in DASH. The code developed in this study has been created by expert programmers in Go, Chapel, Cilk and TBB and can thus be regarded as idiomatic for each approach and free of obvious performance defects.

The five (plus one) problems we consider in our study are the following:

randmat: Generate a (`nrows` $\times$ `ncols`) matrix `mat` of random integers in the range $0, ..., \texttt{max} - 1$ using a deterministic pseudo-random number generator (PRNG) with a given seed value `seed`. The result must be independent of the degree of parallelism (number of threads or processes) employed.

[1] https://bitbucket.org/nanzs/multicore-languages/src

Table 1: Data structures consumed and produced by the Cowichan kernels implemented in this study.

Benchmark	Input	Output
randmat	`nrows`[1], `ncols`[1], `seed`[1], `max`[1]	`mat`[2]
thresh	`mat`[2], `p`[1]	`mask`[2]
winnow	`mat`[2], `mask`[2], `nelem`[1]	`points`[3]
outer	`points`[3]	`omat`[4], `vec`[5]
matvec	`omat`[4], `vec`[5]	`res`[5]
chain	`nrows`[1], `ncols`[1], `seed`[1], `max`[1], `p`[1], `nelem`[1]	`res`[5]

[1] Integer scalar.
[2] (`nrows` $\times$ `ncols`) integer/boolean matrix.
[3] Vector of (row, col) pairs of length `nelem`.
[4] (`nelem` $\times$ `nelem`) matrix of floating point values.
[5] Vector of floating point values of length `nelem`.

thresh: Given an integer matrix `mat`, and a thresholding percentage `p`, compute a boolean matrix `mask` of similar size, such that `mask` selects `p` percent of the largest values of `mat`.

winnow: Given an integer matrix `mat`, a boolean matrix `mask`, and a desired number of target elements `nelem`, perform a *weighted point selection* operation. Apply the mask to the matrix to extract a list of all selected points as 3-tuples in the form (row, col, val), where (row, col) is the location of the selected point in `mat` and val is its value. Sort the list in ascending order by value and select `nelem` equally spaced elements. The result is a vector `points` of (row, col) pairs of length `nelem`.

outer: Given a vector of `nelem` (row, col) points, compute a (`nelem` $\times$ `nelem`) *outer product* matrix `omat` and a vector `vec` of floating point values. For $i \neq j$ the (i, j) entry in `omat` is the Euclidean distance between the i-th point and the j-th point in the list of given points. The diagonals are set to the sum of all entries in a row, times the number of columns. The resulting matrix is symmetric and diagonally dominant. The vector `vec` is formed by computing the Euclidean distance between each point and $(0, 0)$.

matvec: Given an `nelem` $\times$ `nelem` matrix `mat` and a vector `vec`, compute the matrix-vector product (row-by-row inner product) `res`.

chain: Combine the kernels in a sequence such that the output of one becomes the input for the next. I.e., chain = randmat $\circ$ thresh $\circ$ winnow $\circ$ outer $\circ$ matvec.

3 IMPLEMENTATION CHALLENGES AND DASH FEATURES USED

In this section we describe the challenges we encountered when implementing the Cowichan problems. We describe the DASH features used in our code and how they compare to the constructs employed by the reference implementations developed by expert programmers in Go, Chapel, Cilk, and

TBB in the study by Nanz et al. [17]. Naturally this small set of benchmarks only exercises a limited set of the features offered by either programming approach. However, we believe that the requirements embedded in the Cowichan problems are relevant to a wide set of other uses cases, including the classic HPC application areas and beyond.

3.1 Memory Allocation and Data Structure Instantiation

The Cowichan problems use one- and two-dimensional arrays as the main data structures. Table 1 shows the data structures consumed and produced in each of the kernels, referring to the variable names used in most implementation source files [17, 22]). Of course 1D arrays are widely supported by all programming systems. True multidimensional arrays, however, are not universally available and as a consequence workarounds are commonly used. The Cilk and TBB implementation both adopt a linearized representation of the 2D matrix and use a single malloc call to allocate the whole matrix:

```
int *matrix = (int*) malloc(sizeof(int)*nrows*ncols);
```

Access is then not available by using the more natural 2D interface but by explicitly computing the offset of the element in the linearized representation:

```
int val = matrix[i*ncols + j]; // element at (i,j)
```

Go uses a similar approach but bundles the dimensions together with the allocated memory in a custom type:

```
type ByteMatrix struct {
  Rows, Cols int
  array []byte
}
matrix := ByteMatrix{r, c, make([]byte, r*c)}
```

Chapel has a very concise and elegant syntax for the allocation and direct element-wise access of multidimensional arrays:

```
var matrix: [1..nrows, 1..ncols] int(32);
int val = matrix[i, j];
```

The DASH syntax is similarly concise and elegant and offers the natural 2D access interface (using round parentheses, because in C++ the square bracket operator can only take one parameter).

```
dash::NArray<int, 2> matrix(nrows, ncols);
int val = matrix(i,j);
```

3.2 Work Sharing

In all benchmarks work has to be distributed onto multiple processes or threads, for example when computing the random values in *randmat* in parallel. *randmat* requires that the result be independent of the degree of parallelism used and all implementations solve this issue by using a separate deterministic seed value for each row of the matrix. A whole matrix row is then the unit of work that is distributed among the processes or threads. The same strategy is also used for *outer* and *product*.

In Cilk, the corresponding code looks like the following, iteration space partitioning is done automatically by the compiler.

```
cilk_for (int i = 0; i < nrows; i++) {
  // perform operation on row i...
}
```

TBB uses C++ template mechanisms to achieve a similar goal. A custom runtime scheduler manages the distribution of the specified range `(0, nrows)` onto the available threads and for each partition, the given lambda is called, which determines the actual bounds for the current partition and invokes a sequential loop.

```
// TBB
parallel_for(
  // range is typedef for tbb::blocked_range<size_t>
  range(0, nrows),
  [=](range r) {
    auto end = r.end ();
    for (size_t i = r.begin(); i != end; ++i) {
      // perform operation on row i
    }
});
```

Go does not have built-in constructs for simple work sharing loops and the functionality has to be created manually using goroutines, channels, and ranges.

```
// Go
work := make(chan int, 256)

go func() {
  for i := 0; i < nelts; i++ {
    work <- i
  }
  close(work)
}()

done := make(chan bool)
NP := runtime.GOMAXPROCS(0)

// Go (continued)
for i := 0; i < NP; i++ {
  go func() {
    for i := range work {
      // perform operation on row i
    }
    done <- true
  }()
}

for i := 0; i < NP; i++ {
  <-done
}
```

First, the channel `work` is created with a buffer capacity of 256 elements. Then a goroutine is used to populate the channel with all row indices. For each processor in the system a goroutine is then started concurrently that reads values from the channel, the construct `for i := range work` does this automatically until the channel is closed. Finally termination is signaled using the channel `done`.

In Chapel this kind of work distribution can simply be expressed as a parallel loop (`forall`) and the compiler and runtime system have the duty to figure out the actual work distribution.

```
// Chapel
```

```
const rows = 1 .. nrows;
forall i in rows {
  // perform operation on row i
}
```

In DASH, the work distribution follows the data distribution. I.e., each unit is responsible for computing on the data that is locally resident, the owner computes model. This is the corresponding code in DASH:

```
// DASH
auto local = matrix.local;
for (auto i=0; i<local.extent(0); i++) {
  // perform operation on row i
}
```

Each unit determines its locally stored portion of the matrix (guaranteed to be a set of rows by the data distribution pattern used) and works on it independently. In this example the matrix is accessed using local row indices. If the global row index is needed, the container's data distribution pattern can be queried:

```
auto glob = matrix.pattern().global({i,0});
int grow  = glob[0];  // global row of local (i,0)
```

3.3 Global Max Reduction

In *thresh*, the max value held by the matrix has to be determined to initialize other data structures to their correct size. However, the Go reference implementation doesn't actually perform this step and instead just uses a default size of 100, Go is thus not discussed further in this section.

In Cilk the following function, using a `reducer_max` object together with a parallel loop over the rows, is employed to find the maximum. Surprisingly more than ten lines of code are used for this relatively simple task. Evidently, the main complexity stems from the fact that finding the local maximum and combining the local maxima into a global maximum using the `reducer_max` object are handled manually and separately.

```
// Cilk
int reduce_max (int nrows, int ncols) {
  cilk::reducer_max <int> max_reducer (0);

  cilk_for (int i = 0; i < nrows; i++) {
    int begin = i;
    int tmp_max = 0;
    for (int j = 0; j < ncols; j++) {
      tmp_max = std::max (tmp_max, matrix [begin*
          ncols + j]);
    }
    max_reducer.calc_max (tmp_max);
  }

  return max_reducer.get_value ();
}
```

In TBB the following code fragment is used to find the maximum (the code is simplified slightly to remove interwoven code fragments that compute the histogram in the same step). Again a special construct of the programming system is used for performing a parallel reduction (`tbb::parallel_reduce`). The computation of the local maximum and reducing the local values to the global max are again handled separately. The latter step is performed by the `tbb::parallel_reduce` using the perhaps slightly confusing syntax expecting two lambda expressions.

```
// TBB
nmax = tbb::parallel_reduce(
  range(0, nrows), 0,
  [=](range r, int result)->int {
    for (size_t i = r.begin(); i != r.end(); i++) {
      for (int j = 0; j < ncols; j++) {
        result = max(result, (int)matrix[i*ncols + j
            ]);
      }
    }
    return result;
  },
  [](int x, int y)->int {
    return max(x, y);
  });
```

Chapel again has the most concise syntax of all approaches, simply stating the intent of performing a max reduction over the matrix:

```
// Chapel
var nmax = max reduce matrix;
```

The code in DASH is almost as compact, however, since we can use the `max_element()` algorithm to find the maximum. Instead of specifying the matrix object directly, in DASH we have to couple the algorithm and the container using the iterator interface:

```
// DASH
int nmax = (int)*dash::max_element(mat.begin(),
                                   mat.end());
```

Note that this algorithm is not just provided in DASH for this special use cases but is available by *design*. The STL provides `min|max|minmax_element()` as an important and natural algorithmic building block and so does DASH in a generalized parallel variant.

3.4 Parallel Histogramming

thresh requires that a global cumulative histogram over an integer matrix is computed. Thus, for each integer value $0, \ldots, \texttt{nmax} - 1$ we need to determine the number of occurrences in the given matrix in parallel. There are several possible strategies for implementing this. One could keep just a single shared histogram and atomically update it from multiple threads. Alternatively, one or multiple histograms can be computed by each thread in parallel and later combined into a single global histogram. The latter strategy is the one used by all implementations.
Chapel uses an approach where a separate histogram per matrix row is computed and these `nrows` histograms are then combined (in parallel over the 'columns').

```
// Chapel
var histogram: [1..nrows, 0..99] int;
const RowSpace = {1..nrows};
const ColSpace = {1..ncols};

forall i in RowSpace {
  for j in ColSpace {
    histogram[i, matrix[i, j]] += 1;
  }
}

const RowSpace2 = {2..nrows};

forall j in 0..(nmax) {
  for i in RowSpace2 {
    histogram[1, j] += histogram[i, j];
  }
}
```

TBB uses a similar strategy to Chapel's (not shown), whereas Cilk only computes a single histogram per thread, as shown below:

```
// Cilk
void fill_histogram(int nrows, int ncols) {
  int P = __cilkrts_get_nworkers();
  cilk_for (int r = 0; r < nrows; ++r) {
    int Self = __cilkrts_get_worker_number();
    for (int i = 0; i < ncols; i++) {
      histogram [Self][randmat_matrix[r*ncols +i]]++;
    }
  }
}

void merge_histogram () {
  int P = __cilkrts_get_nworkers();
  cilk_for (int v = 0; v < 100; ++v) {
    int merge_val = __sec_reduce_add (histogram [1:(P
        -1)][v]);
    histogram [0][v] += merge_val;
  }
}
```

`__sec_reduce_add` is a bultin function for array section that returns the sum of all array elements.
Go (not shown) uses a similar strategy to Cilk (one histogram per thread) but actually sends the histogram data using a go channel and combines the received histograms in a sequential manner.

In DASH we use a global array to compute the histogram. First, each unit computes the histogram for the locally stored data, by simply iterating over all local matrix elements and updating the local histogram (`histo.local`). Then `dash::transform` is used to combine the local histograms into a single global histogram located at unit 0. `dash::transform` is modeled after `std::transform`, a mutating sequence algorithm. Like the STL variant, the algorithm works with two input ranges that are combined using the specified operation (last parameter) into an output range. The first two iterators are the begin and end of the first input range, the third iterator is the begin of the second input range, and the fourth iterator specifies the output range. In DASH these operations (addition in this example) are guaranteed to be performed atomically and the DASH runtime system uses `MPI_Accumulate`, which guarantees atomic element-wise updates.

```
// DASH
dash::Array<unsigned> hist((nmax + 1) * dash::size())
    ;
dash::fill(hist.begin(), hist.end(), 0);

for(auto ptr = mat.lbegin(); ptr != mat.lend(); ++ptr
    )
  ++(hist.local[*ptr]);

dash::barrier();

if (dash::myid() != 0) {
  dash::transform<unsigned>(hist.lbegin(), hist.lend
      (),
                            hist.begin(), hist.begin
                                (),
                            dash::plus<unsigned>());
}
```

4 EVALUATION

In this section we evaluate DASH in relation to four established parallel programming approaches: Go, Chapel, Cilk and TBB.

Chapel [5] is an object-oriented partitioned global address space (PGAS) programming language developed since the early 2000s by Cray, originally as part of DARPA's High Productivity Computing Systems (HPCS) program. Chapel tries to offer higher-level programming abstractions than what is available from the most commonly used programming approaches in HPC, MPI and OpenMP. We have used Chapel version 1.15.0 in all our experiments.

Go [6] is a general purpose systems-level programming language developed at Google in the late 2000s that focuses on concurrency as a first-class concern. Go supports lightweight threads called *goroutines* which are invoked by prefixing a function call with the 'go' keyword. Channels provide the idiomatic way for communication between goroutines but since all goroutines share a single address space, pointers can also be used for data sharing. We have used Go version 1.8 in our experiments.

Cilk [3] started as an academic project at MIT in the 1990s. Since the late 2000s the technology has been extended and integrated as Cilk Plus into the commercial compiler offerings from Intel and more recently open source implementations for the GNU Compiler Collection (GCC) and LLVM became available. Cilk's initial focus was on lightweight tasks invoked using the *spawn* keyword and dynamic workstealing. Later a parallel loop construct (cilk_for) was added. We have used Cilk as integrated in Intel C/C++ compilers version 17.0.2.174.

Intel Threading Building Blocks (TBB) [20] is a C++ template library for parallel programming that provides tasks, parallel algorithms and containers using a work-stealing approach that was inspired by the early work on Cilk. We have used TBB version 2017.0 in our experiments, which is part of Intel Parallel Studio XE 2017.

DASH (described in more detail in Sect. 2.1) is a realization of the PGAS programming model in the form of a C++ template library. DASH offers distributed data structures and parallel algorithms that conceptually follow the design of the Standard Template Library (STL). DASH programs can be run both on shared memory and on distributed memory systems and one-sided communication primitives are used when communication between nodes is necessary. We used DASH version `dash-0.2.0-2656-ga76ff38` in all experiments which is available as free open source software under a BSD licenese[2].

We investigate different problem sizes of the Cowichan kernels and denote the size used in the form N=`size` to signify the size of the data structures used. In more detail, the following settings are used: `nrows=size`, `ncols=size`, `nelem=size`, `p=50`, `s=31337`. I.e., the seed value for the random number generator is 31337, *thresh* selects 50 percent of the largest values, the size of all matrices is ($\texttt{size} \times \texttt{size}$), and `max` is set to 100 in all instances. These values were chosen for compatibility with the study by Nanz et al, in which all experiments were performed with N=40000. All reported results are the best achieved runtimes in ten repetitions.

4.1 Platforms

The platforms we have used in our evaluation study are as follows:

- **IBEX:** A moderately parallel shared memory multicore system with two sockets using two Intel E5-2630Lv2 (Ivy Bridge-EP) CPUs, 2×6 physical cores, 15 MB L3 cache per CPU, and 64 GB of main memory. The system runs CentOS Linux release 7.2.1511, Linux kernel version 3.10.0-327.
- **KNL:** A modern manycore shared memory system using a Intel Xeon Phi 7210 CPU with 64 cores, Processor base frequency, 32 MB L2 cache (shared among all cores). The system runs SLES 12 SP2, Linux kernel version 4.4.38-93.
- **SuperMUC-WM (multi-node):** Up to 20 'fat' nodes of SuperMUC Phase I (Westmere-EX) hosting four E7-4870 CPUs, 4×10 physical cores, 30 MB L3 cache per CPU, and 256 GB of memory per node. SuperMUC runs SUSE Linux Enterprise Server 11 SP4, Linux kernel version 3.0.101-84.

4.2 Comparison on a Multicore and Manycore System

We first investigate the performance differences between DASH and the four established parallel programming models on two different shared memory systems: IBEX and KNL. The results are shown in Table 2. For Cilk and Chapel it was not possible to run larger problem sizes because these approaches delivered segmentation faults or out-of-memory errors. There was no obvious reason for this problem visible in the benchmark kernels – no stack or other temporary user-visible allocation of data structures could be identified that would causes such a problem. The best performance is generally delivered by TBB but also Chapel shows good performance numbers in our experiments. For *winnow* the largest performance differences of all kernels can be observed. This has less to do with differences in the programming paradigms, however and more with different algorithmic implementation variants. Relatively speaking, TBB and DASH can manage the transition from the moderately parallel IBEX system to the manycore KNL system best.

4.3 Scaling Study

Next we investigate the scaling of the DASH implementation on up to 20 nodes of SuperMUC-HW. Unfortunately, none of the other approaches can be compared with DASH in this scenario. Cilk and TBB are naturally restricted to shared memory systems by their threading-based nature. Go realizes the CSP (communicating sequential processes) model that would, in principle, allow for a distributed memory implementation but since data sharing via pointers is allowed, Go is restricted to a single shared memory node. Finally, Chapel targets both shared and distributed memory systems, but the implementation of the Cowichan problems we use in this study is not prepared to be used with multiple locales and cannot make use of multiple nodes (it lacks the `dmapped` specificion for data distribution).

The runtime results (in seconds) are shown in Table 3 for two problem sizes, N=40000 and N=70000. A speedup plot based on this data is shown in Fig. 2. Naturally for the larger problem size the scaling is better but even the smaller problem scales well up to 5 nodes (200 cores), afterwards the communication requirements dominate the smaller and smaller fraction of computation per core. *product* and *winnow* are the two worst scaling applications due to the high communication requirements for both.

4.4 Productivity

Finally, we evaluate programmer productivity by analyzing source code complexity. Table 4 shows the lines of code (LOC) used in the implementation for each kernel, counting only lines that are not empty or comments. Of course, LOC is a very crude approximation for source code complexity but few other metrics are universally accepted or available for different programming languages. LOC at least gives a rough idea for source code size, and, as a proxy, development time, likelihood for programming errors and productivity. For most kernels, DASH manages to achieve a remarkably low source code size. Keep in mind that of all listed approaches DASH is the only one able to scale beyond a single shared memory node. As a consequence DASH has to take care of data distribution and locality, which is no concern for the other approaches. The reason why DASH still manages to keep a very compact source code size lies in the high-level abstractions offered. First, DASH offers multidimensional array as fundamental data containers and doesn't require the developer to perform manual offset calculations. Second, DASH offers algorithmic building blocks that allow developers to easily express operations that would otherwise be cumbersome in a distributed memory setting.

5 RELATED WORK

C++ libraries implementing the PGAS model for parallel programming similar to DASH are subject to extensive research, in projects such as UPC++[24], Hierarchically Tiled

[2]https://github.com/dash-project/dash

Table 2: Performance results (in seconds runtime) comparing our implementation of the Cowichan problems with existing solutions in go, Chapel, TBB, Cilk on IBEX and KNL with two different problem sizes (N=40000 and N=60000). oom=out of memory.

	IBEX, 12 cores, N=40000					IBEX, 12 cores, N=60000				
	DASH	go	Chapel	TBB	Cilk	DASH	go	Chapel	TBB	Cilk
randmat	0.67	0.68	0.40	0.53	0.56	1.19	1.40	oom	1.13	oom
thresh	0.89	0.99	0.73	2.16	0.89	2.02	2.45	oom	4.73	oom
winnow	7.60	156.84	196.47	2.04	0.84	15.74	392.20	oom	4.58	oom
outer	1.15	1.58	0.82	0.67	0.87	2.58	3.65	oom	1.70	oom
product	0.35	0.50	0.19	0.29	0.28	0.77	1.01	oom	0.59	oom
	KNL, 64 cores, N=40000					KNL, 64 cores, N=60000				
	DASH	go	Chapel	TBB	Cilk	DASH	go	Chapel	TBB	Cilk
randmat	1.54	2.10	0.45	0.516	oom	3.57	3.47	oom	1.10	oom
thresh	0.73	2.73	0.76	1.441	oom	1.67	4.51	oom	2.96	oom
winnow	12.12	782.37	536.76	1.003	oom	27.84	1836.45	oom	2.15	oom
outer	3.99	2.52	1.49	0.865	oom	8.83	6.04	oom	1.94	oom
product	2.34	0.32	0.24	0.262	oom	5.26	0.72	oom	0.59	oom

Table 3: Strong scaling study (runtime in seconds) of the DASH implementation of the Cowichan problems on up to 800 cores of SuperMUC-WM.

	SuperMUC-WM, N=40000					SuperMUC-WM, N=70000				
nodes/cores→	1/40	2/80	5/200	10/400	20/800	1/40	2/80	5/200	10/400	20/800
randmat	0.175	0.087	0.044	0.023	0.014	0.683	0.342	0.142	0.075	0.044
thresh	0.290	0.152	0.070	0.053	0.044	1.111	0.569	0.241	0.135	0.093
winnow	4.826	2.647	1.662	2.310	7.550	13.630	7.130	3.943	4.458	9.990
outer	4.413	2.661	0.887	0.677	0.813	15.842	8.125	4.263	2.038	1.891
product	2.624	1.759	0.763	0.664	0.836	10.505	8.762	2.882	2.627	1.797

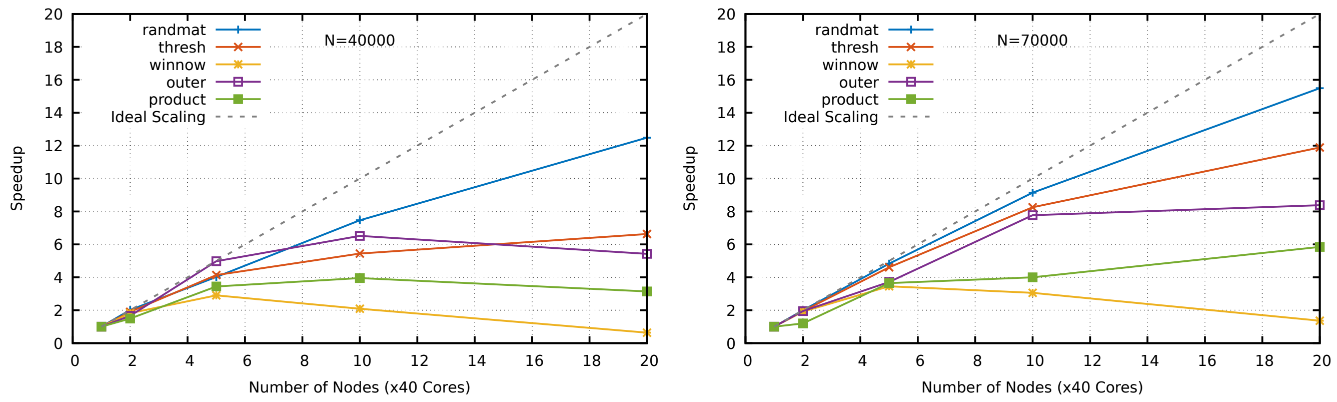

Figure 2: Strong scaling study of the DASH implementation of the Cowichan problems on SuperMUC-WM. Two problem sizes are tested (N=40000 and N=70000) on up to 800 cores (20 nodes).

Arrays [2], C++ Coarrays, STAPL [4], Charm++ [11], and HPX [10].

UPC++ implements a PGAS language model and, similar to the array concept in DASH, offers local views for distributed arrays for rectangular index domains [12]. The UPC++ array library provides a multidimensional array [24], however it currently does not allow the distribution of array elements over multiple processes. In addition, the array abstraction of UPC++ is not compatible with concepts defined in the C++ Standard Template Library (STL) and existing algorithms designed for C++ standard library containers cannot directly be applied to UPC++ arrays.

The Global Arrays (GA) toolkit [18] is dedicated to distributed arrays. Its programming interface is very low-level

Table 4: Lines-of-code (LOC) measure for each kernel and programming approach, counting non-empty and non-comment lines only.

	DASH	go	Chapel	TBB	Cilk
randmat	18	29	14	15	12
thresh	31	63	30	56	52
winnow	67	94	31	74	78
outer	23	38	15	19	15
product	19	27	11	14	10

compared to the DASH solution. The HPX runtime system realizes a language model in C++ that is comparable to PGAS and also shares many design principles with DASH. It does, however, presently not provide support for multidimensional data.

Another approach to PGAS models is followed by XcalableMP (XMP), which is a directives-based language extension for Fortran and C/C++ [13]. The productivity of XMP has been evaluated on the HPC challenge benchmarks with very good results [16].

Our work presented here directly builds upon the study of Nanz et al. [17], who have investigated the implementation of a subset of the Cowichan problems in Cilk, TBB, Chapel and Go. Their objective was to determine the usability and performance potential of these relatively new multicore programming approaches based on development time, source code size, scalability and raw performance. Their study includes source code variants developed by a novice which were improved upon by high-ranking experts for each programming system (including the creators and technical leads for Chapel and TBB). This *expert-parallel* version of the implementation was the basis for our comparisons. The results of their study indicate that Chapel was always by far the most concise language used, however performance was consistently worse by almost an order of magnitude. TBB was the approach with the fastest development time and TBB and Cilk generally delivered the best performance.

Investigating the usability and productivity of parallel programming systems remains a challenging topic. The overview publications by Marowka et al. [14] and Sadowski et al. [21] provide a good introduction to the state of the art and the difficulties involved. Numerous issues such as the selection of the human subjects (programmers), the platforms, and the problem sets make it difficult to derive meaningful conclusions.

The Cowichan problems were specifically developed to enable the determination of usability of parallel programming approaches and they exist in two variants. The first variant [22] describes seven medium-sized problems exhibiting data and task parallelism as well as regular and irregular communication patterns mimicking real applications. The second variant [23] are 13 smaller 'toy' problems that are meant to be quick to implement individually and composable. By combining the problems a wide range of parallel operations are exercises. The study of Nanz et al. and our own work use five problems from the second set of the Cowichan problems.

The work of Paudel et al. [19] and Anvik et al. [1] implement a subset of the first type of Cowichan problems in X10 and CO_2P_3S, respectively. The authors use the problems to investigate the expressiveness of the respective programming system but no qualitative or quantitative comparison with other implementations is made in either study.

6 CONCLUSION

In this paper we have evaluated DASH, a new realization of the PGAS approach in the form of a C++ template library by comparing our implementation of the Cowichan problems with those developed by expert programmers in Cilk, TBB, Chapel, and Go. We show that DASH achieves both remarkable performance and productivity that is comparable with established shared memory programming approaches. DASH is also the only approach in our study where the same source code can be used both on shared memory systems and on scalable distributed memory systems. This step, from shared memory to distributed memory systems is often the most difficult for parallel programmers because it frequently goes hand in hand with a re-structuring of the entire data distribution layout of the application. With DASH the same application can seamlessly scale from a single shared memory node to multiple interconnecting nodes.

Future work is planned in several areas. First, a larger set of benchmark kernels and implementation alternatives would allow for a more thorough and compassing comparison of the various programming approaches. Concretely we plan to develop implementations of the Cowichan problems in OpenMP and MPI to establish baseline performance and productivity numbers using these widely used industry standards. Second, our current implementation of the Cowichan problems does not yet utilize the full potential of more advanced features offered by DASH (such as asynchronous communication) and we plan to continue using the Cowichan problems as test applications for the development of DASH.

REFERENCES

[1] John Anvik, Jonathan Schaeffer, Duane Szafron, and Kai Tan. 2005. Asserting the utility of CO2P3S using the Cowichan Problem Set. *J. Parallel and Distrib. Comput.* 65, 12 (2005), 1542–1557.
[2] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B Fraguela, María J Garzarán, David Padua, and Christoph Von Praun. 2006. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 48–57.
[3] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. 1995. *Cilk: An efficient multithreaded runtime system*. Vol. 30. ACM.
[4] Antal Buss, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M Amato, Lawrence Rauchwerger, et al. 2010. STAPL: standard template adaptive parallel library. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*. ACM, 14.
[5] Bradford L Chamberlain, David Callahan, and Hans P Zima. 2007. Parallel programmability and the Chapel language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312.

[6] Alan A.A. Donovan and Brian W. Kernighan. 2015. *The Go Programming Language* (1st ed.). Addison-Wesley Professional.

[7] Tarek El-Ghazawi and Lauren Smith. 2006. UPC: Unified Parallel C. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 27.

[8] Karl Fürlinger, Tobias Fuchs, and Roger Kowalewski. 2016. DASH: A C++ PGAS Library for Distributed Data Structures and Parallel Algorithms. In *Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications (HPCC 2016)*. Sydney, Australia, 983–990. https://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0140

[9] Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood. 2015. Remote memory access programming in MPI-3. *ACM Transactions on Parallel Computing* 2, 2 (2015), 9.

[10] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. HPX: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 6.

[11] Laxmikant V Kale and Sanjeev Krishnan. 1993. *CHARM++: a portable concurrent object oriented system based on C++*. Vol. 28. ACM.

[12] Amir Kamil, Yili Zheng, and Katherine Yelick. 2014. A local-view array library for partitioned global address space C++ programs. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ACM, 26.

[13] J. Lee and M. Sato. 2010. Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems. In *2010 39th International Conference on Parallel Processing Workshops*. https://doi.org/10.1109/ICPPW.2010.62

[14] Ami Marowka. 2013. Towards Standardization of Measuring the Usability of Parallel Languages. In *International Conference on Parallel Processing and Applied Mathematics*. Springer, 65–74.

[15] MPI Forum. 2012. MPI: A message-passing interface standard. Version 3.0. (Nov. 2012).

[16] Masahiro Nakao, Jinpil Lee, Taisuke Boku, and Mitsuhisa Sato. 2012. Productivity and performance of global-view programming with XcalableMP PGAS language. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE Computer Society, 402–409.

[17] Sebastian Nanz, Scott West, Kaue Soares Da Silveira, and Bertrand Meyer. 2013. Benchmarking usability and performance of multicore languages. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*. IEEE, 183–192.

[18] Jaroslaw Nieplocha, Robert J Harrison, and Richard J Littlefield. 1994. Global Arrays: a portable shared-memory programming model for distributed memory computers. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*. 340–349.

[19] Jeeva Paudel and José Nelson Amaral. 2011. Using the Cowichan problems to investigate the programmability of X10 programming system. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*. ACM, 4.

[20] James Reinders. 2007. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.".

[21] Caitlin Sadowski and Andrew Shewmaker. 2010. The last mile: parallel programming and usability. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 309–314.

[22] Gregory V Wilson. 1994. Assessing the usability of parallel programming systems: The Cowichan problems. In *Programming Environments for Massively Parallel Distributed Systems*. Springer, 183–193.

[23] Gregory V Wilson and R Bruce Irvin. 1995. *Assessing and comparing the usability of parallel programming systems*. University of Toronto. Computer Systems Research Institute.

[24] Yili Zheng, Amir Kamil, Michael B Driscoll, Hongzhang Shan, and Katherine Yelick. 2014. UPC++: a PGAS extension for C++. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 1105–1114.

[25] Huan Zhou, Yousri Mhedheb, Kamran Idrees, Colin Glass, José Gracia, Karl Fürlinger, and Jie Tao. 2014. DART-MPI: An MPI-based Implementation of a PGAS Runtime System. In *The 8th International Conference on Partitioned Global Address Space Programming Models (PGAS)*. https://doi.org/10.1145/2676870.2676875