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

Presented by Joseph Schuchart, Hochleistungsrechenzentrum Stuttgart (HLRS)
Background

Cowichan problems

- A **benchmark suite** designed to investigate the usability of parallel programming systems (1990s)
- Named after a tribal area in the Canadian Northwest
- **1st variant** [1]: 7 medium-sized problems, data and task parallelism, regular and irregular communication patterns
- **2nd variant** [2]: 13 smaller “toy” problems, quick to implement, composable by chaining

---


Cowichan Tribal Area
Chaining the Cowichan Problems (2\textsuperscript{nd} Variant)


- Previous work by Nanz et al. [3] selected five benchmarks to evaluate the usability of multicore languages (2013)

- Four programming systems compared: Go, Cilk, TBB, Chapel

- Metrics:
  - \textbf{Usability}: LOC, development time
  - \textbf{Performance}: execution time and scalability

Example Results from the Study of Nanz et al.

- Comparison of code developed by expert developers

- Chapel has consistently the smallest code size, but worst performance

- Achieving Performance and productivity is not easy

Overview of this work

- We implemented the five benchmarks in DASH
  - DASH is a realization of the PGAS programming model in the form of a C++ template library
  - Offers distributed data structures (e.g., dash::Array) and parallel algorithms (e.g., dash::min_element())

- We compare the DASH implementation with the expert variants\(^1\) from the study of Nanz et al.
  - Comparison of the source code size (LOC)
  - Achieved performance on shared memory systems
  - Scalability study of the DASH implementation on a distributed memory system

\(^1\) Available online: https://bitbucket.org/nanzs/multicore-languages/src
Random Number Generation (randmat)

- Fill a matrix with small random integers
  - Result must be independent of the degree of parallelism
  - Input: nrows, ncols, seed
  - Output: matrix

Example
- nrows=6, ncols=6

```
  6 5 4 1 4 3
  1 0 5 2 9 8
  6 5 2 3 0 3
  1 0 3 4 5 4
  6 9 4 5 0 9
  1 4 1 6 1 4
```
Thresholding (thresh)

- For a given \( p \), compute a boolean mask, such that \( p \) percent of the largest values in a given matrix of values are selected by the mask
  - Input: matrix of values, thresholding percentage \( p \)
  - Output: boolean mask

Example:
  - \( p=50\% \)

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>4</th>
<th>1</th>
<th>4</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

1. Compute Histogram
2. Find Threshold
3. Apply Threshold 4
Weighted Point Selection (winnow) (1)

- Given matrix and mask, construct a list of all selected points, sort the list, and pick \textit{nelem} equally spaced points
  - Input: matrix, mask, nelem
  - Output: list of \textit{nelem} (row, col) points

Example:
- \textit{nelem} = 5

| 6 5 4 1 4 3 | x x x x x | 6 5 4 4 4 |
| 1 0 5 2 9 8 | x x x x x | 5 9 8 5 9 |
| 6 5 2 3 0 3 | x x x x x | 6 5 4 5 4 |
| 1 0 3 4 5 4 | x x x x x | 6 9 4 5 9 |
| 6 9 4 5 0 9 | x x x x x | 4 6 4 5 9 |
| 1 4 1 6 1 4 | x x x x x | 4(5,5)   |

Extract selected points in the form [val (row, col)]
Weighted Point Selection (winnow) (2)

Select nelem equally spaced elements (2D points)

Sort the \([\text{val (row, col)}]\) triples by \(\text{val}\)

\[
\begin{array}{cccccccc}
6(0,0) & 5(0,1) & 4(0,2) & 4(0,4) & 5(1,2) & 9(1,4) & 8(1,5) & 6(2,0) & 5(2,1) & \ldots & 4(5,5)
\end{array}
\]
Outer Product (outer)

- Take a list of 2D points and compute an outer product
  - Product: euclidean distance between two points
  - Diagonals set to nelts*max of row
  - Input: list of nelts points with their row/col coordinates
  - Output: nelts x nelts matrix
  - Output: distance vector vec (distance from (0,0))

Example:

```
(5,5) (4,2) (4,3) (0,0) (1,5)
```

<table>
<thead>
<tr>
<th></th>
<th>(5,5)</th>
<th>(4,2)</th>
<th>(4,3)</th>
<th>(0,0)</th>
<th>(1,5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5,5)</td>
<td>35.36</td>
<td>3.16</td>
<td>2.24</td>
<td>7.07</td>
<td>4.00</td>
</tr>
<tr>
<td>(4,2)</td>
<td>3.16</td>
<td>22.37</td>
<td>1.00</td>
<td>4.48</td>
<td>4.24</td>
</tr>
<tr>
<td>(4,3)</td>
<td>2.24</td>
<td>1.00</td>
<td>25.00</td>
<td>5.00</td>
<td>3.61</td>
</tr>
<tr>
<td>(0,0)</td>
<td>7.07</td>
<td>4.48</td>
<td>5.00</td>
<td>35.36</td>
<td>5.10</td>
</tr>
<tr>
<td>(1,5)</td>
<td>4.00</td>
<td>4.24</td>
<td>3.61</td>
<td>5.10</td>
<td>25.50</td>
</tr>
</tbody>
</table>

Distance of each point from (0,0)

```
7.07 4.47 5.00 0.00 5.10
```

vec

matrix
Matrix-Vector Product (matvec)

- Given a \textit{nelts} x \textit{nelts} Matrix \textit{mat} and a vector \textit{vec}, compute their product

- Example:

\[
\begin{bmatrix}
7.07 & 4.47 & 5.00 & 0.00 & 5.10 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
35.36 & 3.16 & 2.24 & 7.07 & 4.00 \\
3.16 & 22.37 & 1.00 & 4.48 & 4.24 \\
2.24 & 1.00 & 25.00 & 5.00 & 3.61 \\
7.07 & 4.48 & 5.00 & 35.36 & 5.10 \\
4.00 & 4.24 & 3.61 & 5.10 & 25.50 \\
\end{bmatrix}
\begin{bmatrix}
* \\
\end{bmatrix}
\begin{bmatrix}
295.72 \\
148.99 \\
163.67 \\
121.00 \\
195.29 \\
\end{bmatrix}
\]
## Cowichan Data Structures and Algorithms

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Data Structures</th>
<th>Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>randmat</td>
<td>2D Matrix</td>
<td>Work Sharing</td>
</tr>
<tr>
<td>thresh</td>
<td>2D Matrix</td>
<td>Work Sharing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Global Max Reduction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Global Histogram</td>
</tr>
<tr>
<td>winnow</td>
<td>2D Matrix, 1D Array</td>
<td>Work Sharing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Count If</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sort</td>
</tr>
<tr>
<td>outer</td>
<td>2D Matrix, 1D Array</td>
<td>Work Sharing</td>
</tr>
<tr>
<td>matvec</td>
<td>2D Matrix, 1D Array</td>
<td>Work Sharing</td>
</tr>
</tbody>
</table>
Implementation: Data Structures

- Data structure allocation, element access
  - Mostly 1D and 2D arrays used in all benchmarks

```cilk, tbb
int *matrix = (int*) malloc (sizeof(int)*nrows*ncols);
int val = matrix[i*ncols + j]; // element at (i,j)
```

```go
// Go
type ByteMatrix struct {
    Rows, Cols int
    array [][]byte
}
matrix := ByteMatrix{nrows, ncols, 
    make([][]byte, nrows*ncols)}
```

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

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

- No native support for 2D data organization
  - User-defined data type that represents the 2D matrix
  - Built-in support for working with 2D data
**Implementation: Work Sharing (1)**

- **Work sharing**
  - Unit of work = one matrix row in all implementations

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

```tbb
// 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
        }
    });
```

- Loop parallelism is easily expressed in Cilk and TBB
- Mapped to an internal task-based execution model automatically
Go has no built-in feature for loop-based parallelism

Instead, work sharing is manually implemented using Go channels
Implementation: Work Sharing (3)

- The Chapel implementation uses `forall` to parallelize the loop over the rows.

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

- Work distribution follows data distribution in DASH (owner-computes model).
- `.local` represents the locally stored rows of the matrix.
- If the global row index is needed, the data distribution pattern can be consulted.

```
// DASH
auto local = matrix.local;

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

auto glob = matrix.pattern().global({i,0});
int grow = glob[0]; // global row of local (i,0)
```
Goal: find the largest value in the matrix in parallel

```cilk
// 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 ();
}
```

Cilk uses a reducer_max object to find the maximum value held in each row of the matrix in parallel.
Implementation: Global Max Reduction (2)

// 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*nrows + j]);
            }
        }
        return result;
    },
    [](int x, int y) -> int {
        return max(x, y);
    });

TBB divides the rows among threads, finds the max in each partition and reduces the partial results with the specified comparison function (lambda expression)

// Chapel
var nmax = max reduce matrix;

dash::max_element() uses std::max_element() for the local max.

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

Global reduction yields global max.
Goal: compute a histogram of the values occurring in a matrix in parallel

Chapel starts with a “histogram matrix” (one histogram per row) which is computed in parallel over the matrix rows.

Then the histogram matrix is folded into a single histogram in parallel over the columns.
// 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;
    }
}

- Cilk first computes one histogram per thread (or “worker”)
- Then the histograms are added using the __sec_reduce_add builtin function
DASH first computes the histogram for the local part of the matrix
The local histograms are then combined into a global histogram using 
dash::transform()
Results – Lines of Code

<table>
<thead>
<tr>
<th></th>
<th>DASH</th>
<th>go</th>
<th>Chapel</th>
<th>TBB</th>
<th>Cilk</th>
</tr>
</thead>
<tbody>
<tr>
<td>randmat</td>
<td>18</td>
<td>29</td>
<td>14</td>
<td>15</td>
<td>12</td>
</tr>
<tr>
<td>thresh</td>
<td>31</td>
<td>63</td>
<td>30</td>
<td>56</td>
<td>52</td>
</tr>
<tr>
<td>winnow</td>
<td>67</td>
<td>94</td>
<td>31</td>
<td>74</td>
<td>78</td>
</tr>
<tr>
<td>outer</td>
<td>23</td>
<td>38</td>
<td>15</td>
<td>19</td>
<td>15</td>
</tr>
<tr>
<td>product</td>
<td>19</td>
<td>27</td>
<td>11</td>
<td>14</td>
<td>10</td>
</tr>
</tbody>
</table>

- DASH is not the most concise approach, but not much worse than the best solution
  - DASH is the only case where the same code can be run on shared memory and distributed memory systems!
Hardware (first system)

- IBEX: Two-socketed Ivy Bridge-EP system, 2x6 physical cores, 64 GB of main memory

<table>
<thead>
<tr>
<th></th>
<th>IBEX, 12 cores, N=40000</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DASH</td>
<td>go</td>
<td>Chapel</td>
<td>TBB</td>
</tr>
<tr>
<td>randmat</td>
<td>0.67</td>
<td>0.68</td>
<td>0.40</td>
<td>0.53</td>
</tr>
<tr>
<td>thresh</td>
<td>0.89</td>
<td>0.99</td>
<td>0.73</td>
<td>2.16</td>
</tr>
<tr>
<td>winnow</td>
<td>7.60</td>
<td>156.84</td>
<td>196.47</td>
<td>2.04</td>
</tr>
<tr>
<td>outer</td>
<td>1.15</td>
<td>1.58</td>
<td>0.82</td>
<td>0.67</td>
</tr>
<tr>
<td>product</td>
<td>0.35</td>
<td>0.50</td>
<td>0.19</td>
<td>0.29</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>IBEX, 12 cores, N=60000</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DASH</td>
<td>go</td>
<td>Chapel</td>
<td>TBB</td>
</tr>
<tr>
<td>randmat</td>
<td>1.19</td>
<td>1.40</td>
<td>oom</td>
<td>1.13</td>
</tr>
<tr>
<td>thresh</td>
<td>2.02</td>
<td>2.45</td>
<td>oom</td>
<td>4.73</td>
</tr>
<tr>
<td>winnow</td>
<td>15.74</td>
<td>392.20</td>
<td>oom</td>
<td>4.58</td>
</tr>
<tr>
<td>outer</td>
<td>2.58</td>
<td>3.65</td>
<td>oom</td>
<td>1.70</td>
</tr>
<tr>
<td>product</td>
<td>0.77</td>
<td>1.01</td>
<td>oom</td>
<td>0.59</td>
</tr>
</tbody>
</table>

Results:

- DASH doesn’t consistently achieve the best results, but we’re not that far off
- Chapel and Cilk have issues with their memory management
Results – Shared Memory (2)

- Hardware (second system)
  - KNL: Intel Xeon Phi 7210 CPU with 64 cores

<table>
<thead>
<tr>
<th></th>
<th>KNL, 64 cores, N=40000</th>
<th>KNL, 64 cores, N=60000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DASH</td>
<td>go</td>
</tr>
<tr>
<td>randmat</td>
<td>1.54</td>
<td>2.10</td>
</tr>
<tr>
<td>thresh</td>
<td>0.73</td>
<td>2.73</td>
</tr>
<tr>
<td>winnow</td>
<td>12.12</td>
<td>782.37</td>
</tr>
<tr>
<td>outer</td>
<td>3.99</td>
<td>2.52</td>
</tr>
<tr>
<td>product</td>
<td>2.34</td>
<td>0.32</td>
</tr>
</tbody>
</table>

Results

- DASH doesn’t consistently achieve the best results, but we’re not that far off
- Chapel and Cilk have issues with memory management
- Chapel achieves surprisingly good results (much better than reported by Nanz et al.)
Results – Scaling Study

- Hardware: SuperMUC-WM (Intel Westmere-EX)
  - Up to 20 nodes (4 × 10 physical cores)

Results

- Reasonably good scaling up to 200 cores (N=40000) and 400 cores (N=70000) for most benchmarks
- Winnow is most challenging application (requires sorting)
Conclusion

- We’ve investigated the performance and productivity of DASH in comparison with Go, Cilk, Chapel, TBB using a subset of the Cowichan problems

Results
- On shared memory systems, DASH achieves competitive scores on both productivity and performance
- Additionally the same DASH code scales on distributed memory systems up to moderate parallelism

The productivity in DASH comes from
- Appropriate data structures for the problem domain
- Parallel algorithms
Acknowledgements

- **DASH is on GitHub**
  - [https://github.com/dash-project/dash/](https://github.com/dash-project/dash/)
  - [http://www.dash-project.org/](http://www.dash-project.org/)

- **The DASH Team**
  T. Fuchs (LMU), R. Kowalewski (LMU), D. Hünich (TUD), A. Knüpfer (TUD), J. Gracia (HLRS), C. Glass (HLRS), H. Zhou (HLRS), K. Idrees (HLRS), F. Mößbauer (LMU), J. Schuchart (HLRS), D. Bonilla (HLRS), K. Fürlinger (LMU)

- **Funding**
  [DFG](https://www.dfg.de/)
  [Bundesministerium für Bildung und Forschung](https://www.bmbf.de/)

The Cowichan Problems in DASH