LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

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 parallelsim, regular and irregular communication patterns
  - **2nd variant** [2]: 13 smaller "toy"' problems, quick to implement, composable by chaining

[1] Wilson, Gregory V. *"Assessing the usability of parallel programming systems: The Cowichan problems."* In *Programming Environments for Massively Parallel Distributed Systems*, pp. 183-193. Birkhäuser, Basel, 1994.

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

Cowichan Tribal Area

Chaining the Cowichan Problems (2nd Variant)

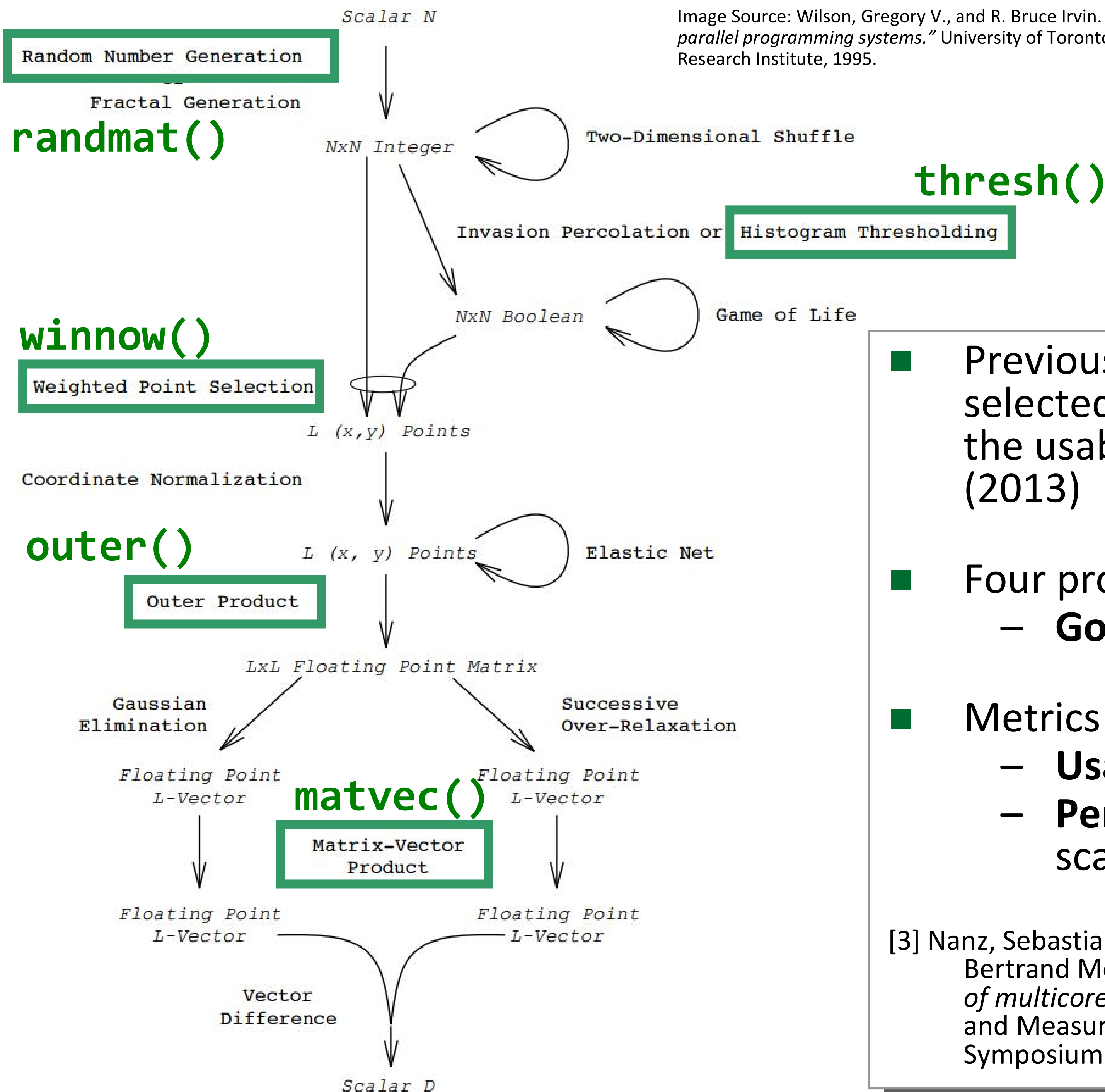


Image Source: Wilson, Gregory V., and R. Bruce Irvin. *"Assessing and comparing the usability of parallel programming systems."* University of Toronto. Computer Systems Research Institute, 1995.

- Previous work by Nanz et al. [3] selected **five bechmarks** to evaluate the usability of multicore languages (2013)
- Four programming systems compared:
  - **Go**, **Cilk**, **TBB**, **Chapel**
- Metrics:
  - **Usability**: LOC, development time
  - **Performance**: execution time and scalability

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

Example Results from the Study of Nanz et al.

- Comparison of code developed by **expert** developers

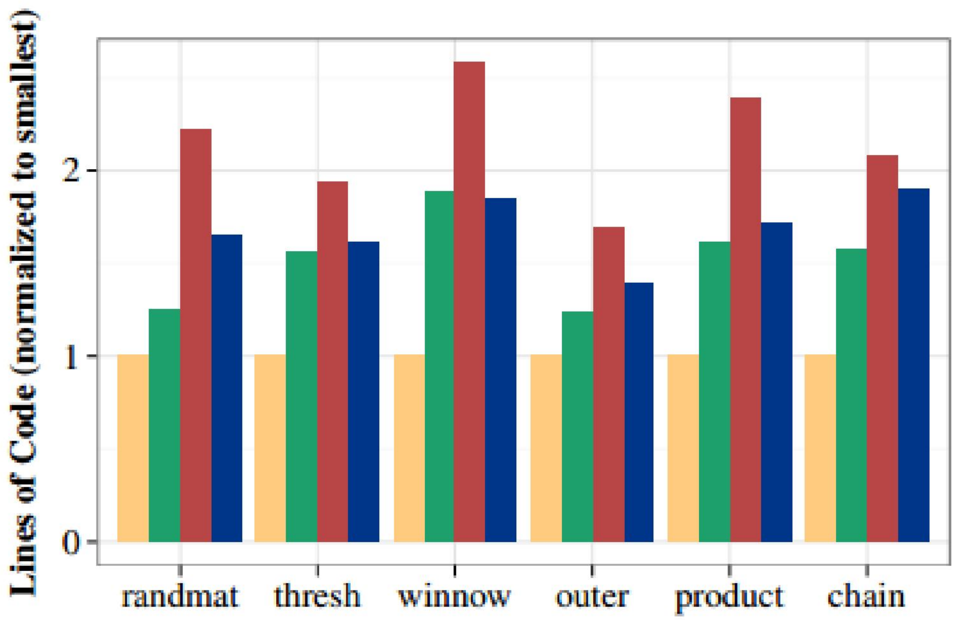


Figure 1. Source code size (LoC)

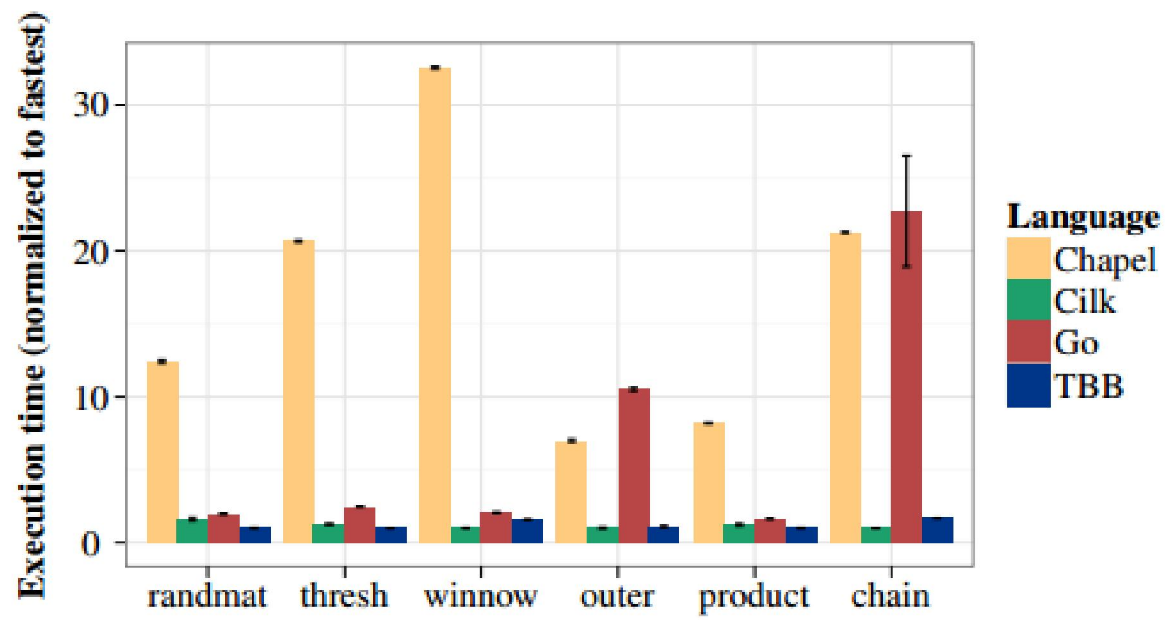


Figure 5. Execution time

Source: Nanz, Sebastian, Scott West, Kaue Soares Da Silveira, and Bertrand Meyer. *"Benchmarking usability and performance of multicore languages."* In Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on, pp. 183-192. IEEE, 2013.

- 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.
  - Comparision 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**

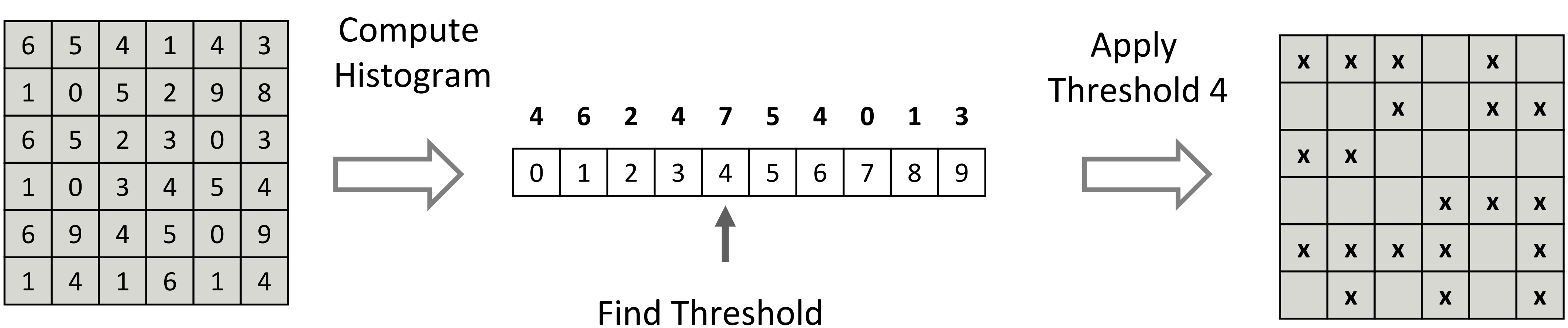
- Example
  - nrows=6, ncols=6

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%



Weighted Point Selection (winnow) (1)

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

- Example:
  - nelem = 5

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

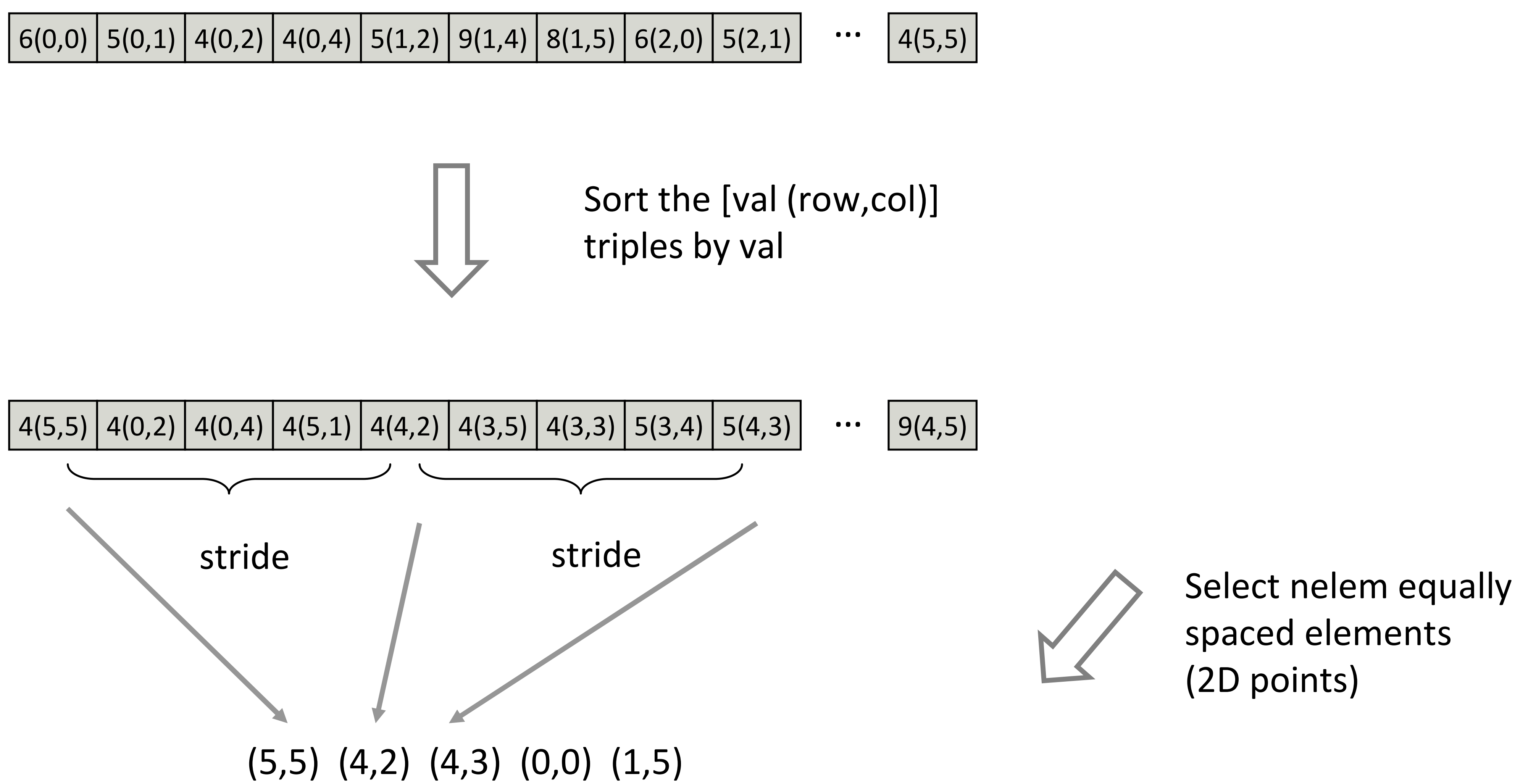
x	x	x		x	
		x		x	x
x	x				
			x	x	x
x	x	x	x		x
	x		x		x

6	**5**	**4**		**4**	
		5		**9**	**8**
6	**5**				
			4	**5**	**4**
6	**9**	**4**	**5**		**9**
	4		**6**		**4**

Extract selected points
in the form [val (row,col)]

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)	…	4(5,5)

Weighted Point Selection (winnow) (2)



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)
(5,5)	35.36	3.16	2.24	7.07	4.00
(4,2)	3.16	22.37	1.00	4.48	4.24
(4,3)	2.24	1.00	25.00	5.00	3.61
(0,0)	7.07	4.48	5.00	35.36	5.10
(1,5)	4.00	4.24	3.61	5.10	25.50

matrix

Distance of each point from (0,0)

7.07	4.47	5.00	0.00	5.10

vec

Matrix-Vector Product (matvec)

- Given a *nelts* x *nelts* Matrix **mat** and a vector **vec**, compute their product

- Example:

7.07	4.47	5.00	0.00	5.10

*

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

=

295.72
148.99
163.67
121.00
195.29

Cowichan Data Structures and Algorithms

Benchmark	Data Structures	Algorithms
randmat	2D Matrix	Work Sharing
thresh	2D Matrix	Work Sharing Global Max Reduction Global Histogram
winnow	2D Matrix 1D Array	Work Sharing Count If Sort
outer	2D Matrix 1D Array	Work Sharing
matvec	2D Matrix 1D Array	Work Sharing

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)
```

- No native support for 2D data organization

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

- User-defined data type that represents the 2D matrix

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

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

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

```
// 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

Implementation: Work Sharing (2)

```go
// 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)

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
}
```

- Go has no built-in feature for loop-based parallelism
- Instead, work sharing is manually implemented using Go channels

Implementation: Work Sharing (3)

```
// Chapel
const rows = 1 .. nrows;

forall i in rows {
   // perform operation on row i
}
```

- The Chapel implemenation uses forall to parallelize the loop over the rows

```
// 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)
```

- 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

Implementation: Global Max Reduction (1)

- Goal: find the largest value in the matrix in parallel

```
// 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*ncols + 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 comparision function (lambda expression)

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

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

- dash::max_element() uses std::max_element() for the local max.
- Global reduction yields global max.

Implementation: Global Historgram (1)

- Goal: compute a histogram of the values occuring in a matrix in parallel

```
// 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];
  }
}
```

- 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

Implementation: Global Historgram (2)

```
// 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

Implementation: Global Historgram (3)

```
// DASH
dash::Array<unsigned int>
      histo((nmax + 1)*dash::size());

dash::fill(histo.begin(), histo.end(), 0);

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

dash::barrier();

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

- 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

	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

- 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!

Results – Shared Memory (1)

- Hardware (first system)
  - IBEX: Two-socked Ivy Bridge-EP system, 2x6 physical cores, 64 GB of main 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

- 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

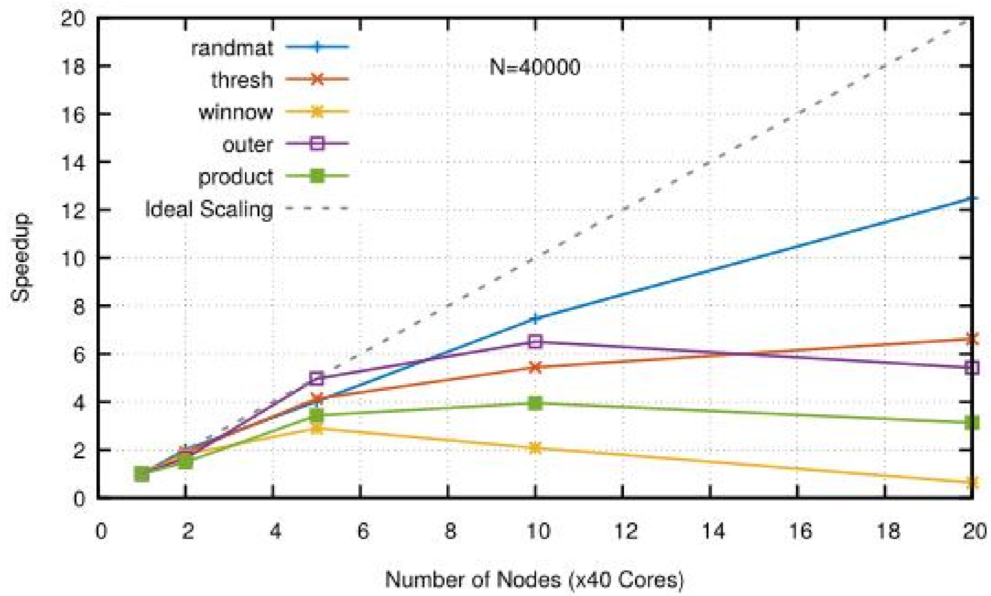
	KNL, 64 cores, N=40000				
	DASH	go	Chapel	TBB	Cilk
randmat	1.54	2.10	0.45	0.516	oom
thresh	0.73	2.73	0.76	1.441	oom
winnow	12.12	782.37	536.76	1.003	oom
outer	3.99	2.52	1.49	0.865	oom
product	2.34	0.32	0.24	0.262	oom

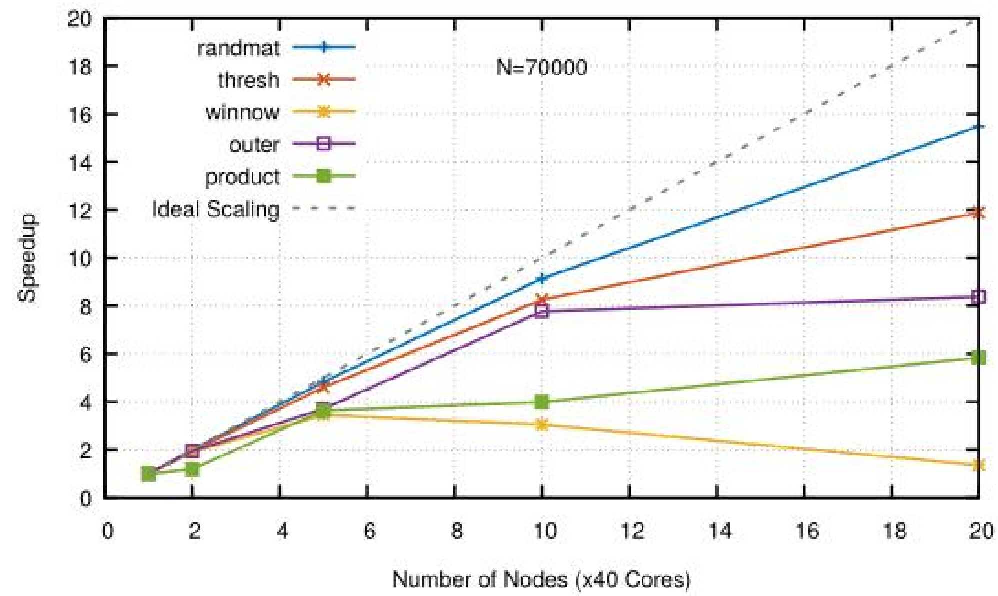
	KNL, 64 cores, N=60000				
	DASH	go	Chapel	TBB	Cilk
randmat	3.57	3.47	oom	1.10	oom
thresh	1.67	4.51	oom	2.96	oom
winnow	27.84	1836.45	oom	2.15	oom
outer	8.83	6.04	oom	1.94	oom
product	5.26	0.72	oom	0.59	oom

- 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 paralleism

- 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/
  - 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 Deutsche Forschungsgemeinschaft

  Bundesministerium für Bildung und Forschung