Concepts and Models of Parallel and Data-centric Programming

Distributed Shared Memory

Lecture, Summer 2020

Dr. Christian Terboven <terboven@itc.rwth-aachen.de>

i12 High Performance Computing | RWTH AACHEN UNIVERSITY

Outline

0. Organization
1. Foundations
2. Shared Memory
3. GPU Programming
4. Bulk-Synchronous Parallelism
5. Message Passing
6. **Distributed Shared Memory**
7. Parallel Algorithms
8. Parallel I/O
9. MapReduce
10. Apache Spark

a. PGAS Foundations
b. DASH Overview
c. Distributed Data Structured
d. DASH Algorithms
e. Tasking

Distributed Data Structures

Distributed Data Structures

- DASH offers distributed data structures
  - Support for flexible data distribution schemes
  - Example: dash::Array<T>

```
dash::Array<int> arr(100);

if( dash::myid()==0 ) {
  for( auto i=0; i<arr.size(); i++ )
    arr[i]=i;
}

arr.barrier();
if(dash::myid()==1 ) {
  for( auto el: arr )
    cout<<(int)el<<" ";
  cout<<endl;
}
```

DASH global array of 100 integers, distributed over all units, default distribution is BLOCKED

Unit 0 writes to the array using the global index i. Operator [] is overloaded for the dash::Array.

Unit 1 executes a range-based for loop over the DASH array

```
$ mpirun -n 4 ./array
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53
54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87
88 89 90 91 92 93 94 95 96 97 98 99
```

Accessing Local Data

- Access to the local portion of the data is exposed through a local-view proxy object (.local)

```
dash::Array<int> arr(100);

for( auto i=0; i<arr.lsize(); i++ )
  arr.local[i]=dash::myid();

arr.barrier();
if(dash::myid()==dash::size()-1 ) {
  for( auto el: arr )
    cout<<(int)el<<" ";
  cout<<endl;
}
```

.lsize() is short hand for .local.size() and returns the number of local elements

.local is a *proxy object* that represents the part of the data that is local to a unit.

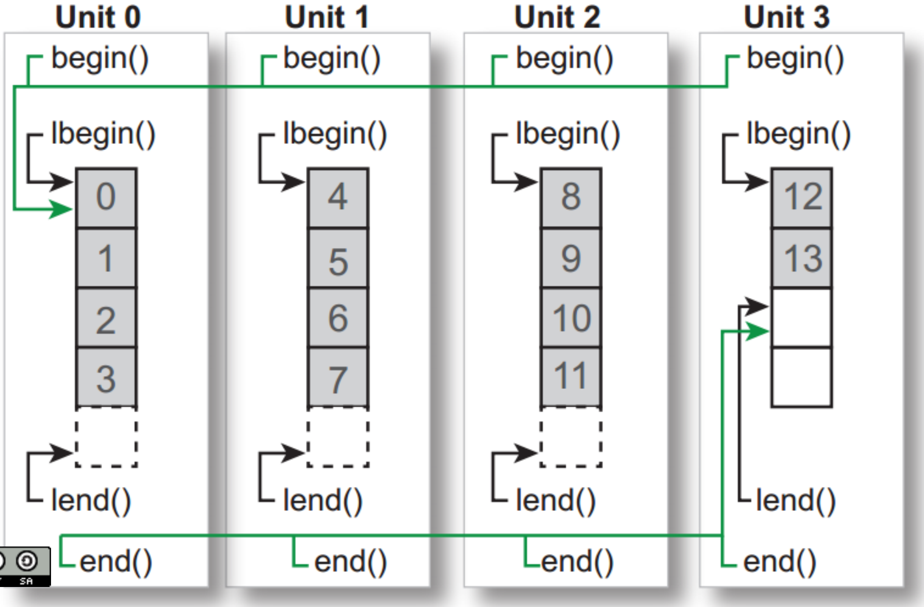
```
$ mpirun -n 4 ./array
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3
```

Global-view and Local-view in DASH

- DASH supports both *global-view* and *local-view* semantics

	Global-view	Local-view	LV shorthand
range begin	arr.begin()	arr.local.begin()	arr.lbegin()
range end	arr.end()	arr.local.end()	arr.lend()
# elements	arr.size()	arr.local.size()	arr.lsize()
element access	arr[glob_idx]	arr.local[loc_idx]	

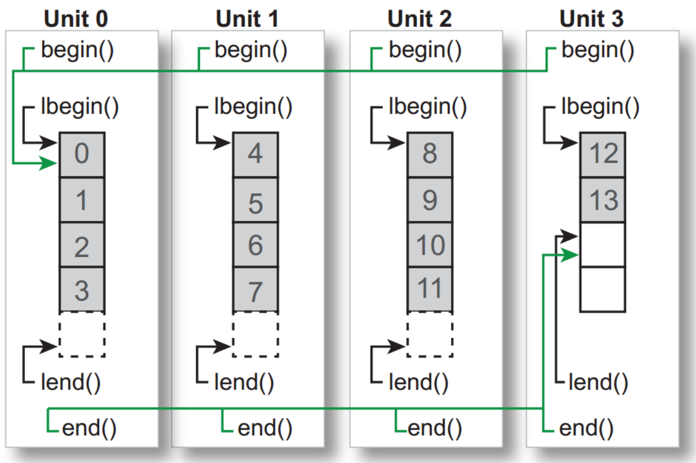
- Example
  - dash::Array with 14 elements, distributed over 4 units
  - default distribution: BLOCKED
  - Blocksize = ceil(14/4)=4



Efficient Local Access (1)

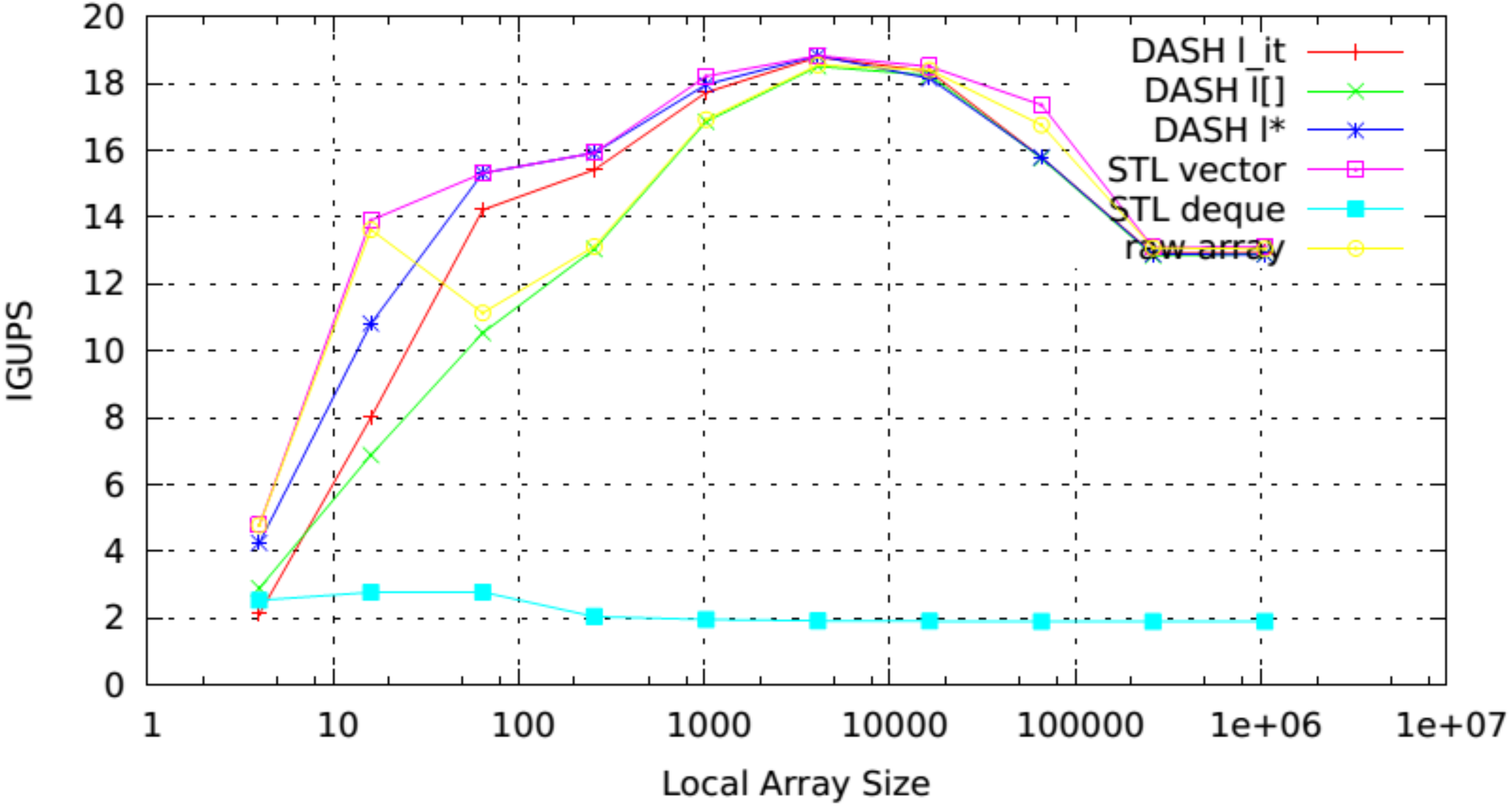
- Several options for access to local data

```
dash::Array<int> arr(1000);

// get raw pointer to local mem.
int *p1 = arr.local.begin();
int *p2 = arr.lbegin(); //p1==p2

// access via local index
arr.local[22] = 33;

// range-based for loop
for( auto el : arr.local )
  cout<<el<<" ";

// access using local iterators
for( auto it=arr.lbegin(); it!=arr.lend(); ++it ) {
  (*it) = foo(...);
}
```



Efficient Local Access (2)

- IGUPs Benchmark: independent parallel updates



Using STL Algorithms

- STL algorithms can be used with DASH containers
  - Both on the local view and the global view

```
#include <libdash.h>

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

  dash::Array<int> a(1000);

  if( dash::myid()==0 ) {
    // global iterators and std. algorithms
    std::sort(a.begin(), a.end());
  }

  // local access using local iterators
  std::fill(a.lbegin(), a.lend(), 23+dash::myid());

  dash::finalize();
}
```

Collective constructor, all units involved

STL algorithms work with DASH global iterators

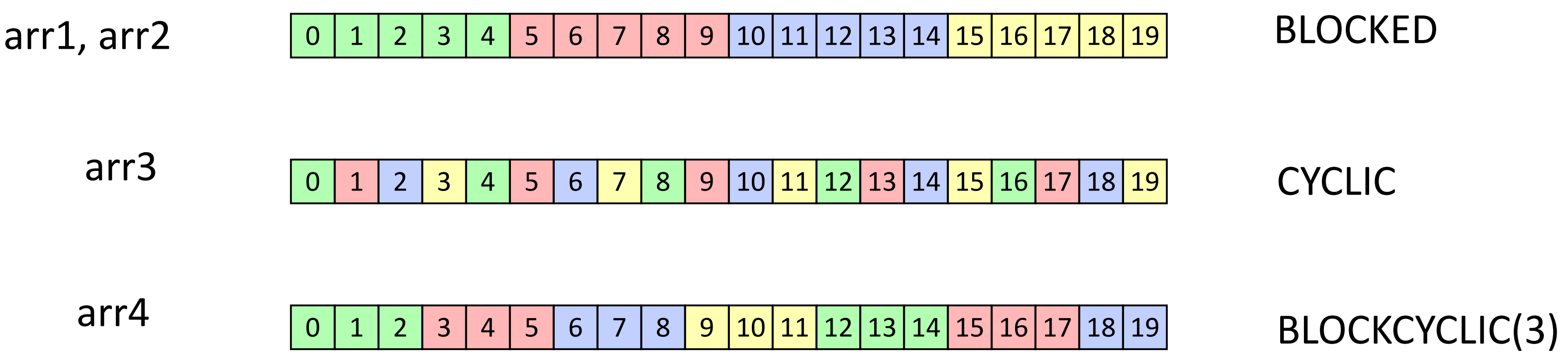
STL algorithms work with DASH local iterators

Data Distribution Patterns

- The data distribution pattern is configurable

```
dash::Array<int> arr1(20); // default: BLOCKED

dash::Array<int> arr2(20, dash::BLOCKED)
dash::Array<int> arr3(20, dash::CYCLIC)
dash::Array<int> arr4(20, dash::BLOCKCYCLIC(3))
```

- Assume 4 units



arr1, arr2	🟩 0	🟩 1	🟩 2	🟩 3	🟩 4	🟥 5	🟥 6	🟥 7	🟥 8	🟥 9	🟦 10	🟦 11	🟦 12	🟦 13	🟦 14	🟨 15	🟨 16	🟨 17	🟨 18	🟨 19	BLOCKED
arr3	🟩 0	🟥 1	🟦 2	🟨 3	🟩 4	🟥 5	🟦 6	🟨 7	🟩 8	🟥 9	🟦 10	🟨 11	🟩 12	🟥 13	🟦 14	🟨 15	🟩 16	🟥 17	🟦 18	🟨 19	CYCLIC
arr4	🟩 0	🟩 1	🟩 2	🟥 3	🟥 4	🟥 5	🟦 6	🟦 7	🟦 8	🟨 9	🟨 10	🟨 11	🟩 12	🟩 13	🟩 14	🟥 15	🟥 16	🟥 17	🟦 18	🟦 19	BLOCKCYCLIC(3)

Accessing Local Data – Cyclic Distribution

- The previous example with a cyclic distribution:

```
// this is the only changed line
dash::Array<int> arr(100, dash::CYCLIC);

for( auto i=0; i<arr.lsize(); i++ )
  arr.local[i]=dash::myid();

arr.barrier();
if(dash::myid()==dash::size()-1 ) {
  for( auto el: arr )
    cout<<(int)el<<" ";
  cout<<endl;
}
```

```
$ mpirun -n 4 ./array
0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0
1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1
2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2
3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
```

DASH Distributed Data Structures Overview

Container	Description	Data distribution
Array<T>	1D Array	static, configurable
NArray<T, N>	N-dim. Array	static, configurable
Shared<T>	Shared scalar	fixed (at 0)
Directory(*)<T>	Variable-size, locally indexed array	manual

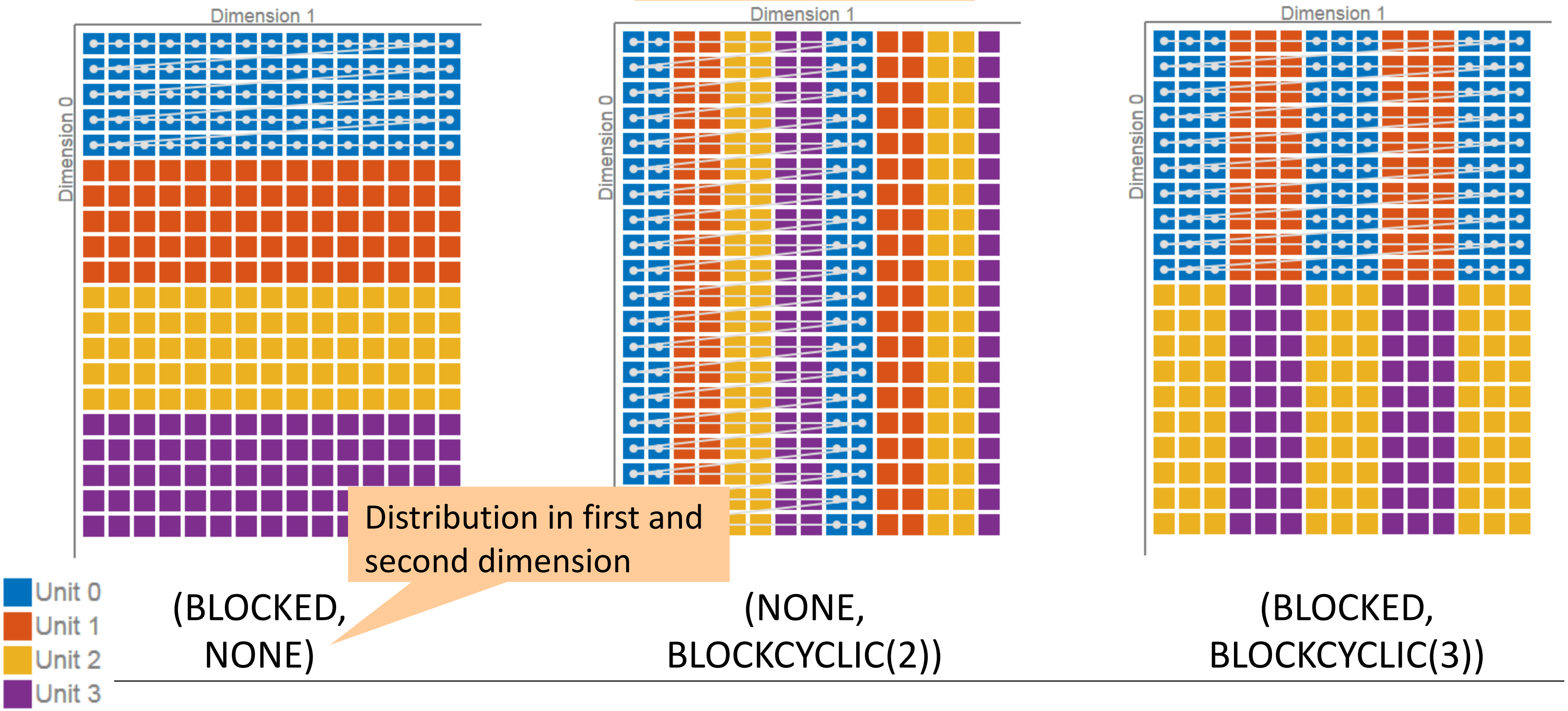
(*) Under development

Multidimensional Data Distribution (1)

- dash::Pattern<N> specifies N-dim data distribution
  - Blocked, cyclic, and block-cyclic in multiple dimensions

Pattern<2>(20, 15)

Extent in first and second dimension



Distribution in first and second dimension

(BLOCKED, NONE)

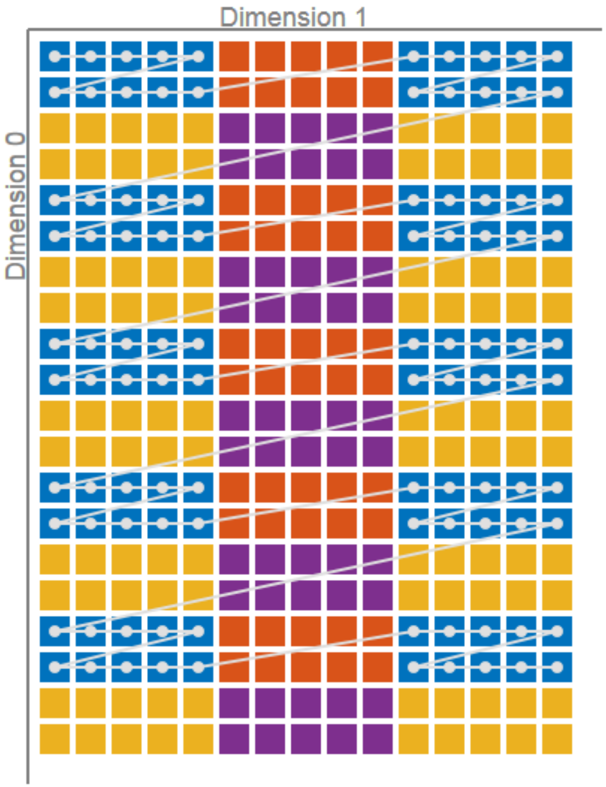
(NONE, BLOCKCYCLIC(2))

(BLOCKED, BLOCKCYCLIC(3))

Multidimensional Data Distribution (2)

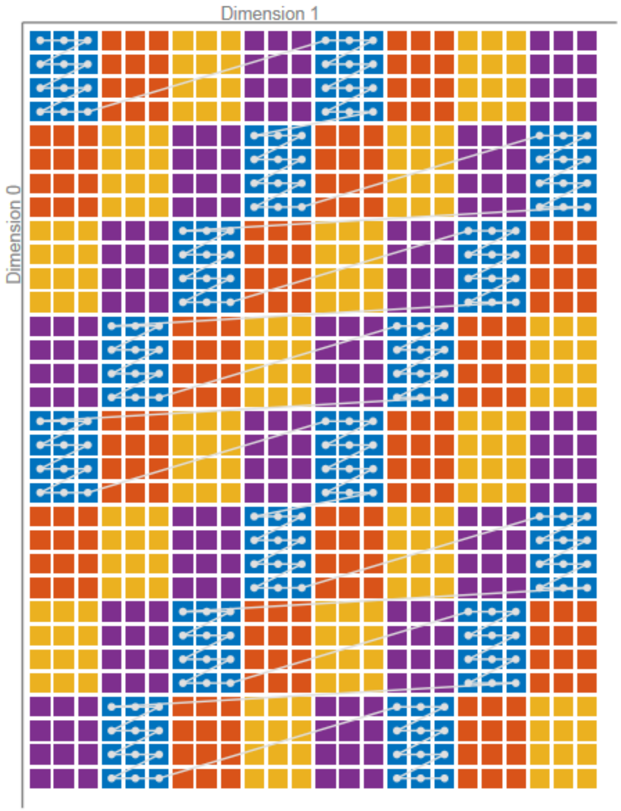
- Tiled data distribution and tile-shifted distribution

TilePattern<2>(20, 15)



(TILE(2), TILE(5))

ShiftTilePattern<2>(32, 24)



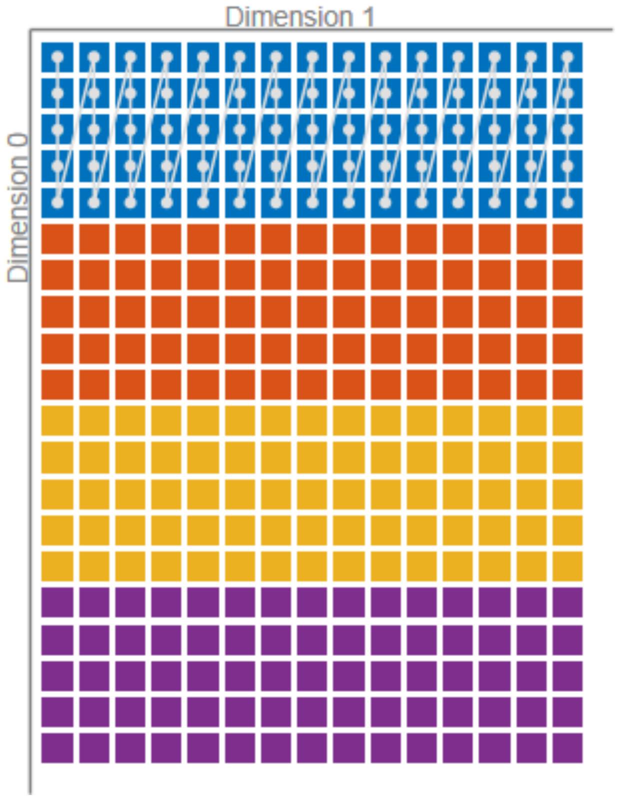
(TILE(4), TILE(3))

🟦 Unit 0
🟥 Unit 1
🟨 Unit 2
🟪 Unit 3

Multidimensional Data Distribution (3)

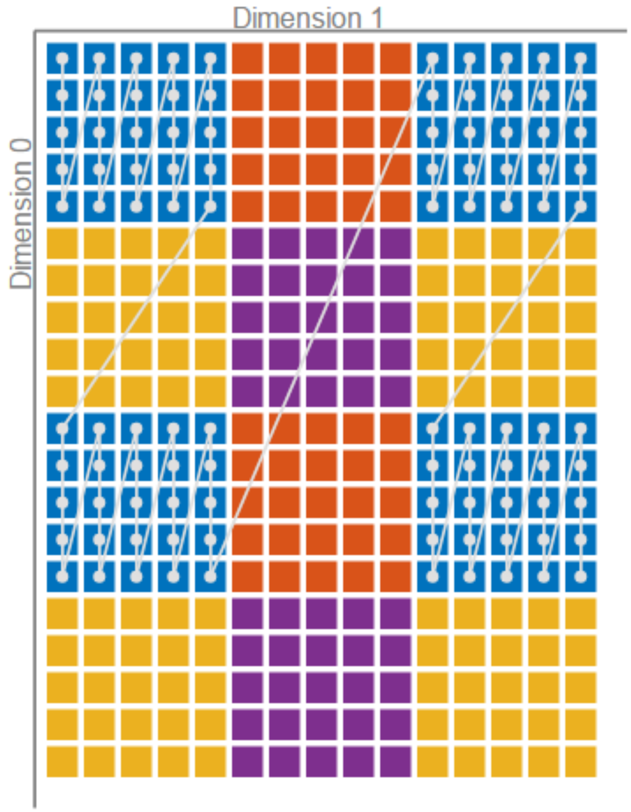
- Row-major and column-major storage

Pattern<2, COL_MAJOR>(20, 15)



(BLOCKED, NONE)

TilePattern<2, COL_MAJOR>(20, 15)



(TILE(5), TILE(5))

🟦 Unit 0
🟥 Unit 1
🟨 Unit 2
🟪 Unit 3

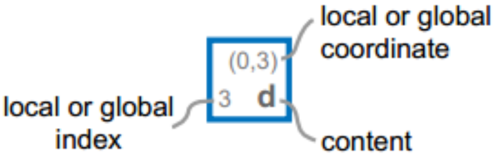
The N-Dimensional Array

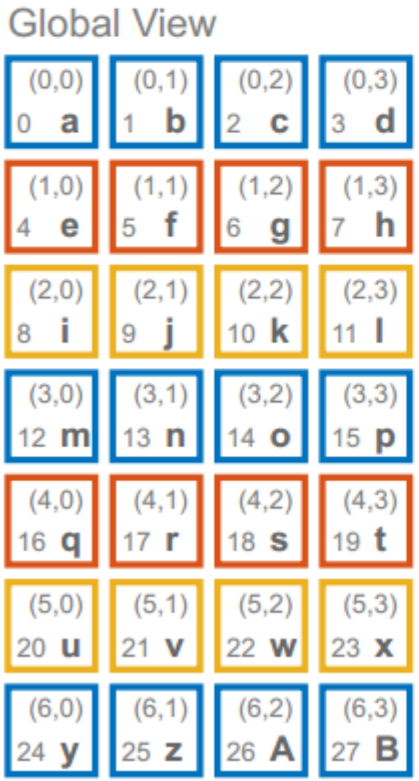
- dash::NArray (dash::Matrix) offers a distributed multidimensional array abstraction
  - Dimension is a template parameter
  - Element access using coordinates or linear index
  - Support for custom index types
  - Support for row-major and column-major storage

```
dash::NArray<int, 2> mat(40, 30); // 1200 elements

int a = mat(i,j);   // Fortran style access
int b = mat[i][j];  // chained subscripts

auto loc = mat.local;

int c = mat.local[i][j];
int d = *(mat.local.begin()); // local iterator
```

DASH NArray Global View and Local View

- Local view works similar to 1D array

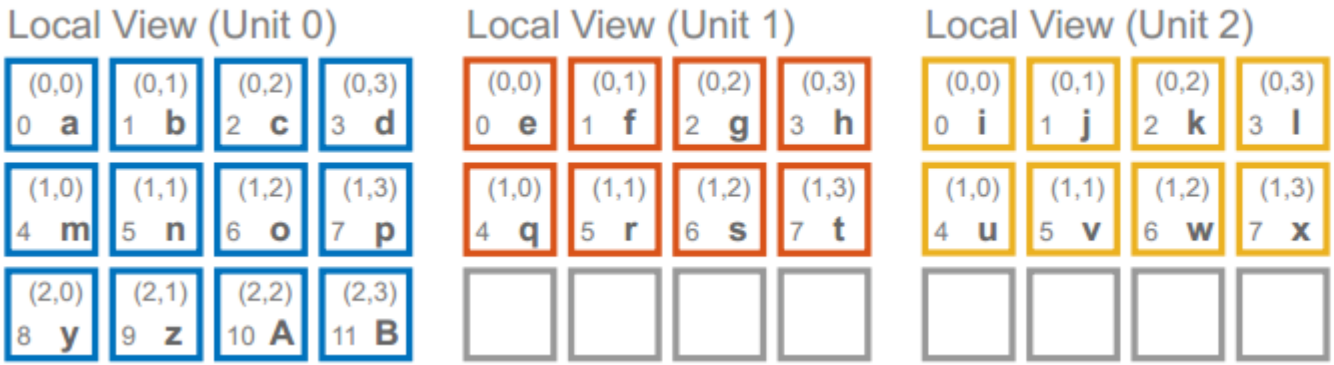




Global View

(0,0) 0 a	(0,1) 1 b	(0,2) 2 c	(0,3) 3 d
(1,0) 4 e	(1,1) 5 f	(1,2) 6 g	(1,3) 7 h
(2,0) 8 i	(2,1) 9 j	(2,2) 10 k	(2,3) 11 l
(3,0) 12 m	(3,1) 13 n	(3,2) 14 o	(3,3) 15 p
(4,0) 16 q	(4,1) 17 r	(4,2) 18 s	(4,3) 19 t
(5,0) 20 u	(5,1) 21 v	(5,2) 22 w	(5,3) 23 x
(6,0) 24 y	(6,1) 25 z	(6,2) 26 A	(6,3) 27 B

```
dash::NArray<char, 2> mat(7, 4);
cout << mat(2, 1) << endl;  // prints 'j'

if(dash::myid()==0) {
  cout << mat.local(2, 1) << endl; // prints 'z'
}
```



Local View (Unit 0)

(0,0) 0 a	(0,1) 1 b	(0,2) 2 c	(0,3) 3 d
(1,0) 4 m	(1,1) 5 n	(1,2) 6 o	(1,3) 7 p
(2,0) 8 y	(2,1) 9 z	(2,2) 10 A	(2,3) 11 B

Local View (Unit 1)

(0,0) 0 e	(0,1) 1 f	(0,2) 2 g	(0,3) 3 h
(1,0) 4 q	(1,1) 5 r	(1,2) 6 s	(1,3) 7 t

Local View (Unit 2)

(0,0) 0 i	(0,1) 1 j	(0,2) 2 k	(0,3) 3 l
(1,0) 4 u	(1,1) 5 v	(1,2) 6 w	(1,3) 7 x