How to Use C++ Parallel Algorithms in an MPI Setup

Armin Sobhani
asobhani@sharcnet.ca

SHARCNET | Compute Canada
HPC Technical Consultant

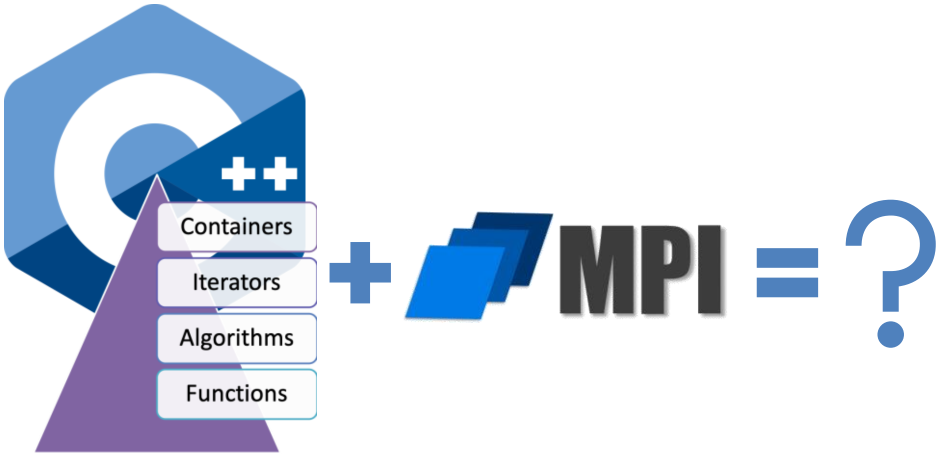
OntarioTech UNIVERSITY

July 29, 2020

computecanada

Compute • Calcul Ontario

SHARCNET™

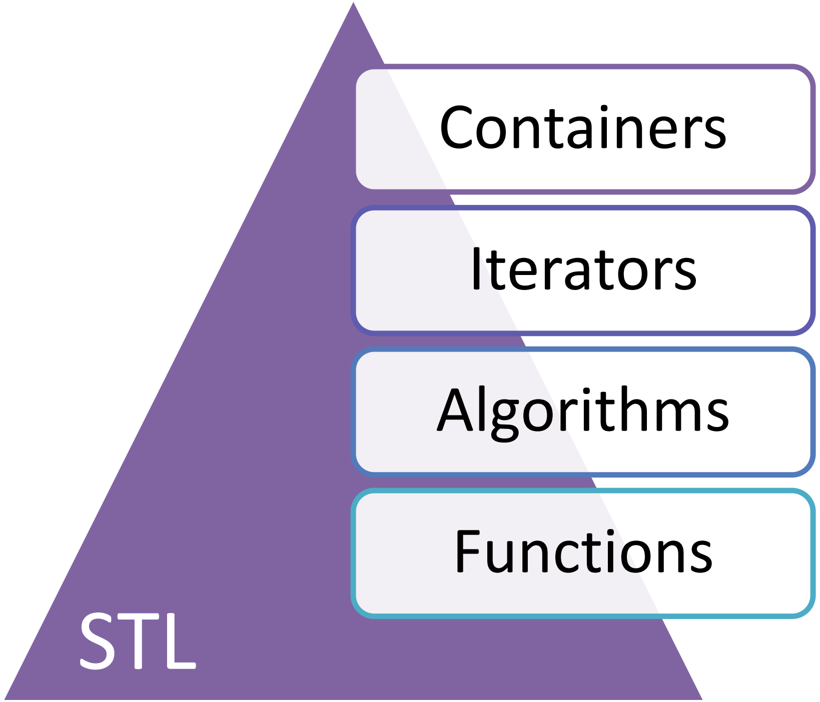


Outline

- A very short intro to *C++17 parallel algorithms*
- An overview of *Partitioned Global Address Space* (PGAS) parallel programming model
- Introducing *DASH C++* template library
- A live demo of installing and building programs with DASH
- Demo project on GitHub: https://github.com/arminms/dash-tutorial

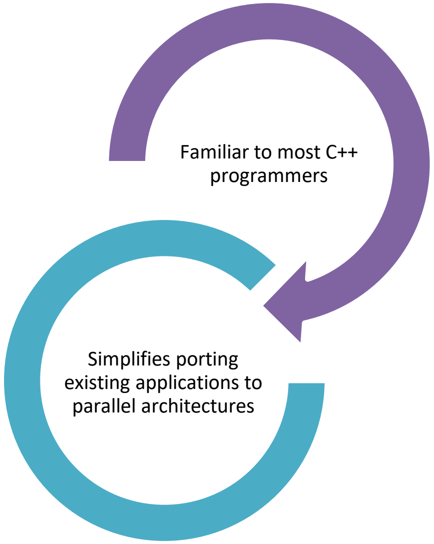
Standard Template Library (STL)

- Software library for the C++
- Influenced many parts of the C++ Standard Library
- Consisting of 4 components:



Parallel STL

Why?



Available Implementations

C++17 Parallel Algorithms

- Microsoft Visual Studio 2017 15.5
- Intel's open source Parallel STL
- STE||AR Group's HPX library
- KhronosGroup's SYCL Parallel STL

Third-Party C++ Libraries

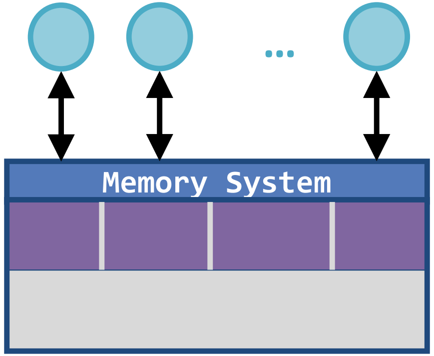
- Boost.Compute
- Thrust by Nvidia
- Bolt by AMD

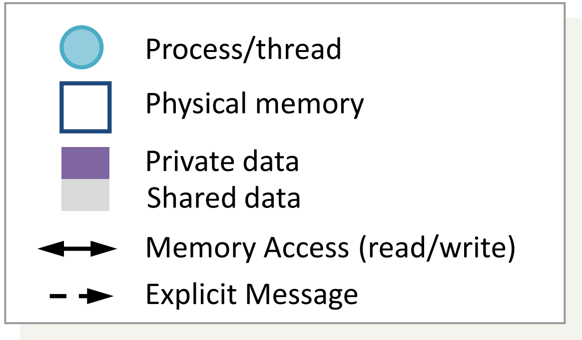
Is There a Parallel Algorithms Implementation for MPI?

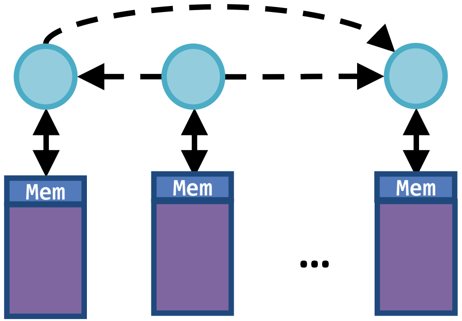
- The straight answer is NO
- Why not?
  - Requires a new type of distributed containers
  - Requires new types of iterators/algorithms that support both local and global iterations, AKA *Affinity*
- But hold on…

Programming Parallel Machines

The two most widely used approaches for parallel programming



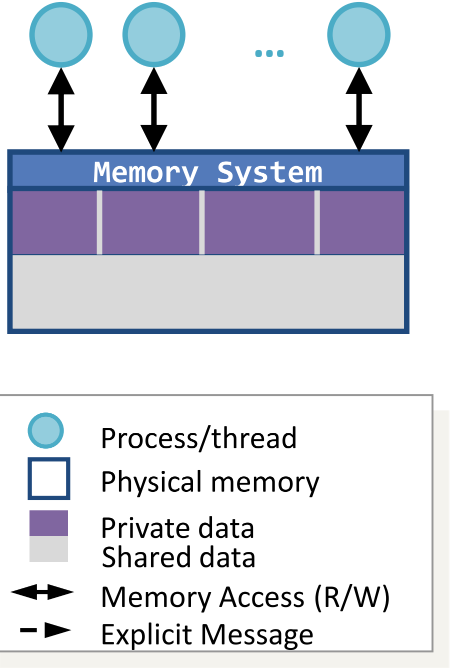




Shared Memory
programming using
Threads

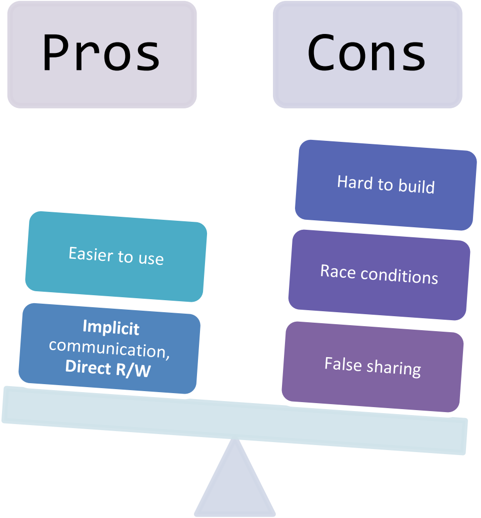
Distributed Memory
programming using
Message Passing (MPI)

Shared Memory Programming using Threads

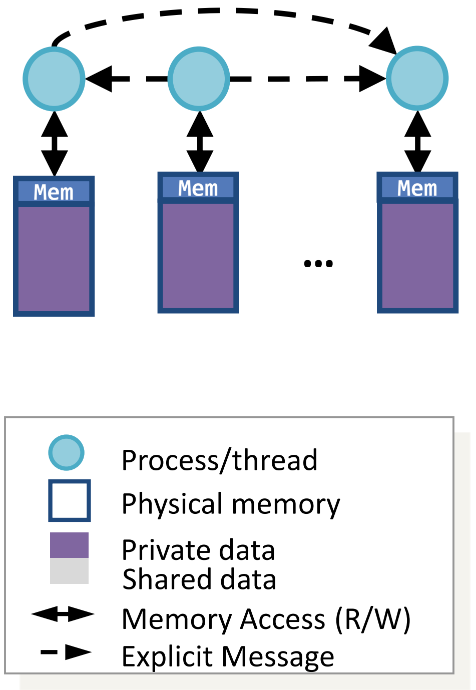


- All the CPU-cores can access the same memory

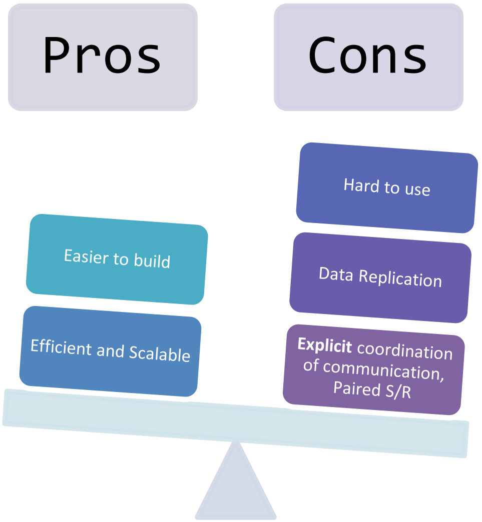
- Examples:
  - OpenMP
  - Pthreads
  - C++ threads
  - Java threads



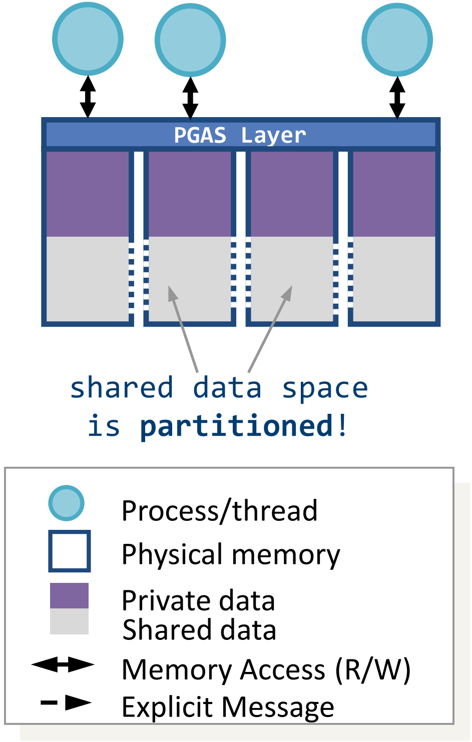
Distributed Memory Programming using Message Passing



- The CPU-cores cannot access the same memory
- Example
  - MPI (Message Passing Interface)



Partitioned Global Address Space (PGAS)



- Best of both worlds
- Can be used on large scale distributed memory as well as shared memory architectures
- A PGAS program looks much like a regular threaded program, but
  - Sharing data is declared **explicitly**
  - The data partitioning is made **explicit**
  - Both needed for performance!

PGAS – Relies on

One-sided Communications in MPI (AKA RDMA or RMA)

Non-Blocking Synchronization

Non-Uniform Memory Access (NUMA)

Cache Only Memory Architecture (COMA)

PGAS – Implementations

Runtime Middleware Layers that Exploit RDMA-Enabled Networks

GASnet

ARMCI

GASPI

OpenShmem

MPI-3 RMA

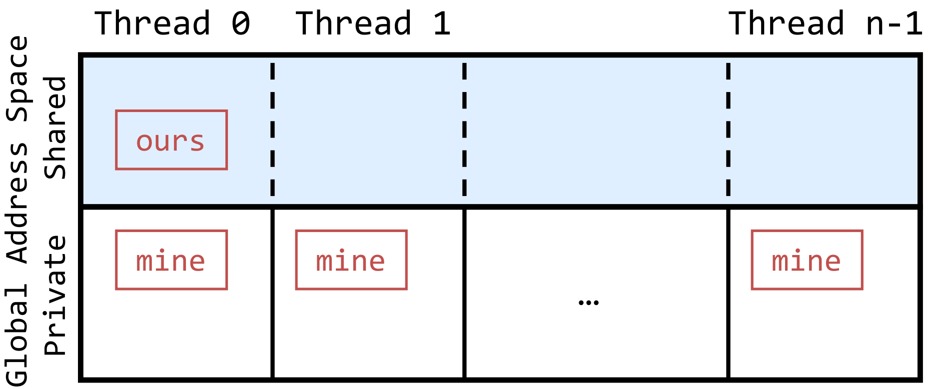
PGAS – Implementations

Language Extensions			Languages			Libraries			
Unified Parallel C (UPC)	Coarray Fortran	Split-C	Sun’s Fortress	Cray’s Chapel	IBM’s X10	UPC++	Coarray C++	SHMEM	DASH

PGAS – How it Works

- An example in **Unified Parallel C** or **UPC**'s terms:
  - Let's call the members of our program **threads**
  - Let's assume we use the **SPMD** (single program multiple data) paradigm
  - Let's assume we have a new keyword "**shared**" that puts variables in the shared global address space

```
shared int ours;
int mine;
```

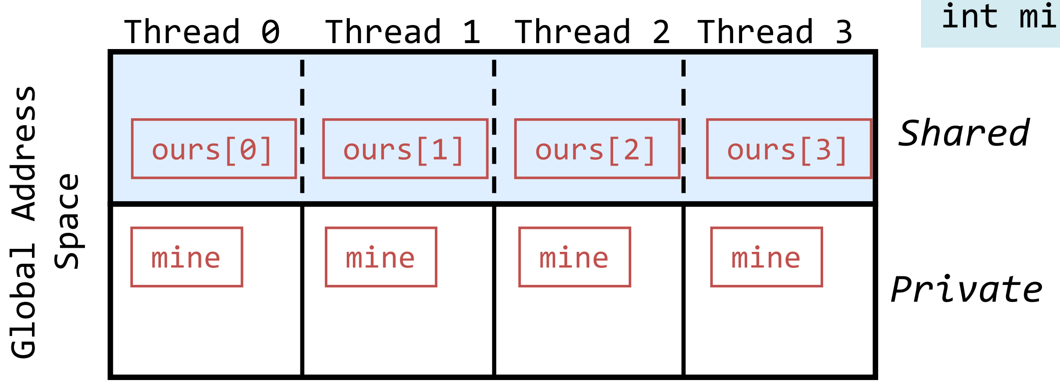


- **1** copy of **ours**
  - Accessible by every thread
- **n** copies of **mine** (one per thread)
  - Each thread can only access its own copy

PGAS – Shared Arrays

- A shared array example in UPC

```
shared int[4] ours;
int mine;
```



- Affinity – in which partition a data item "lives"
  - **`ours`** (previous slide) lives in partition **`0`** (by convention)
  - **`ours[i]`** lives in partition **`i`**

PGAS – Global-View vs. Local-View

- Two ways to organize access to shared data:
  - Global-view e.g. UPC

```
shared int X[100];
X[i]=23;
```

Global size,
Global index

- X is declared in terms of its global size
- X is accessed in terms of global indices
- process (image) is not specified explicitly

  - Local-view e.g. Co-Array Fortran

```
integer :: a(100)[*], b(100)[*]
b(17) = a(17)[2]
```

Local index,
Local size

co-dimension / co-index
in square brackets

- a, b are declared in terms of their local size
- a, b are accessed in terms of local indices
- process (image) is specified explicitly (the co-index)

PGAS – Summary

- PGAS is a concept realized in UPC and other languages and extensions

- UPC, for example, is an extention to C, implementing the PGAS model
  - Built on top of a middleware layer like GASNet
  - Available as a gcc version, Berkeley UPC, from some vendors

- Cons
  - Often not part of the standard software stack of HPC systems
  - Tricky to install and tune for individual users
  - No collective operations or algortithms (e.g. reduce)

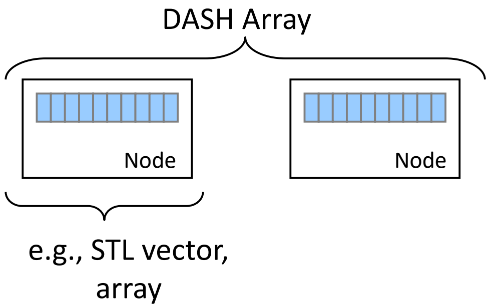
DASH – Overview

- PGAS in the form of a C++ Template library
  - Focus on data structures

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

a[23] = 412;
std::cout << a[42] << std::endl;
```

http://www.dash-project.org/

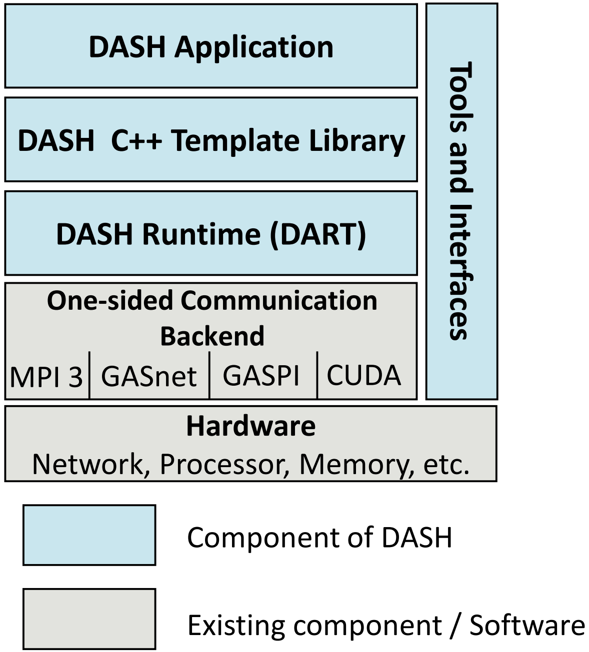
- Array **a** can be stored in the memory of several nodes
- **`a[i]`** transparently refers to local memory or to remote memory via operator overloading



- Not a new language to learn
  - Can be integrated with existing (MPI) applications
  - Relies on MPI3 RMA
- Support for hierarchical locality
  - Team hierarchies and locality iterators

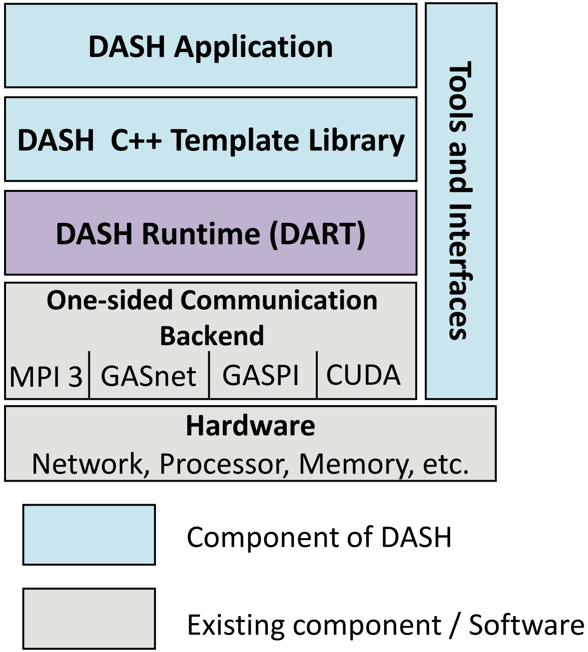
DASH 101

DASH – A C++ Template Library



- A complete **PGAS** programming system without a custom (pre-) compiler
- It offers:
  - Distributed data structures (**containers**) and **parallel algorithms**
    - **STL** conformity / iterator interface
  - **HDF5** input/output

DART – The DASH Runtime Interface



- Plain-C (99) interface
- SPMD execution model
- Defines Units and Teams
- Global memory abstraction
- One-sided RDMA operations
- Several implementations:
  - DART-SHMEM
    Shared-memory based implementation
  - DART-CUDA
    Supports GPUs, based on DART-SHMEM
  - DART-GASPI
    Initial implementation using GASPI
  - DART-MPI
    MPI-3 RDMA "workhorse" implementation

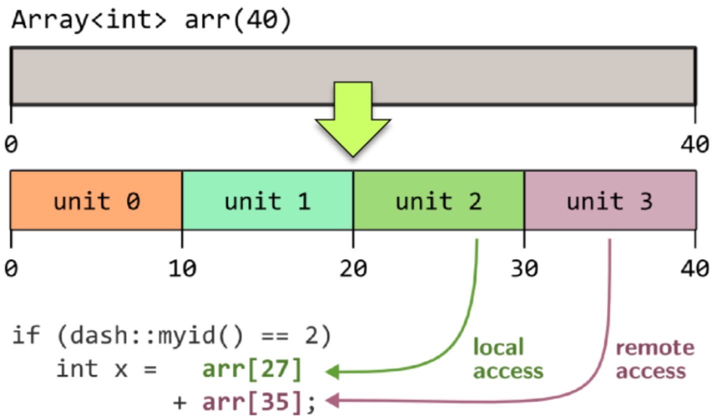
Units and Teams in DART

- *Unit* – individual participants in a DASH/DART program
  - Unit ≈ process (**MPI**) ≈ thread (**UPC**) ≈ image (**CAF**)
  - Execution model follows the classical **SPMD** (Single Program Multiple Data) paradigm
  - Each unit has a *global ID* that remains unchanged during the execution

- *Team*
  - Ordered subset of units
  - Identified by an integer ID
  - DART_TEAM_ALL represents all units in a program
  - Units that are members of a team have a *local ID* with respect to that team

PGAS in DASH

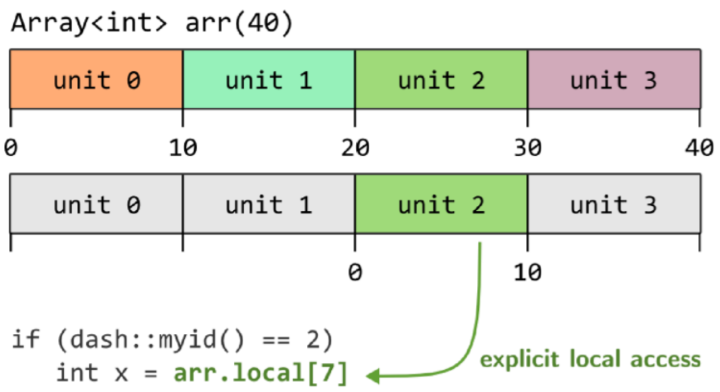
- Data Affinity – data has well-defined owner but can be accessed by any unit



- Unified access to local and remote data in *global* memory space

PGAS in DASH

- Data Affinity – data has well-defined owner but can be accessed by any unit



- Unified access to local and remote data in *global* memory space
- And explicit views on *local* memory space

Hello World in DASH

```
#include <iostream>

#include <libdash.h>

using namespace std;

int main(int argc, char* argv[])
{
    pid_t pid; char buf[100];

    dash::init(&argc, &argv);

    auto myid = dash::myid();
    auto size = dash::size();
    gethostname(buf, 100); pid = getpid();

   cout << "'Hello world' from unit " << myid <<
       " of " << size <<" on " << buf <<
       " pid=" << pid << endl;

   dash::finalize();
}
```

Initialize the programming environment

Determine total number of units and our own unit ID

Print message. Note SPMD model, similar to MPI

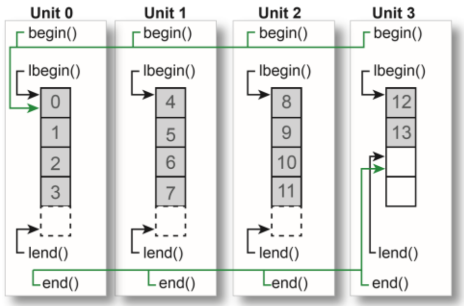
```
$ mpirun -n 4 hello_mpi
'Hello world' from unit 0 of 4 on gra-login2 pid=22872
'Hello world' from unit 1 of 4 on gra-login2 pid=22873
'Hello world' from unit 2 of 4 on gra-login2 pid=22878
'Hello world' from unit 3 of 4 on gra-login2 pid=22879
```

Global-View vs. 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`



Distributed Data Structures

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

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

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

arr.barrier();

if (dash::myid() == 0)
    for (auto val : arr)
        cout << static_cast<int>(val) << " ";

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 ./global_mpi
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);
auto myid = dash::myid();

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

arr.barrier();

if (dash::myid() == 0)
    for (auto val : arr)
        cout << static_cast<int>(val) << " ";

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 ./local_mpi
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
```

Using STL Algorithms

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

```
#include <iostream>

#include <libdash.h>

using namespace std;


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

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

    // local access using local iterators
    std::fill(a.lbegin(), a.lend(), 100 + dash::myid());

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

    dash::finalize();
}
```

Collective constructor, all units involved

STL algorithms work with DASH local iterator ranges

… as well as DASH global iterator ranges

Distributed Data Structures in DASH

Container	Description	Data Distribution
`Array<T>`	1D Array	static, configurable
`NArray<T, N>`	N-dimensional Array	static, configurable
`Shared<T>`	Shared scalar	fixed (at 0)
`Directory<T>`*	Variable-size, locally indexed array	manual, load-balanced
`List<T>`	Variable-size linked list	dynamic, load-balanced
`Map<T>`	Variable-size associative map	dynamic, balanced by hash function

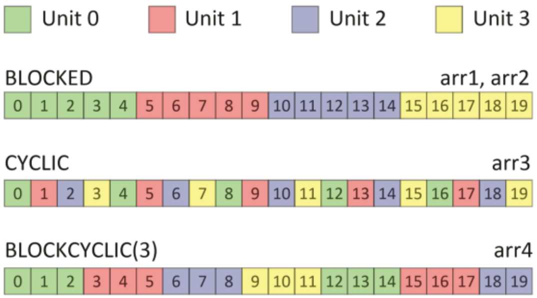
* Under construction

Data Distribution Patterns

- Data distribution patterns are configurable in DASH

```
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))
// use your own data distribution:
dash::Array<int, MyPattern> arr5(20, MyPattern(…))
```

- Four units layout



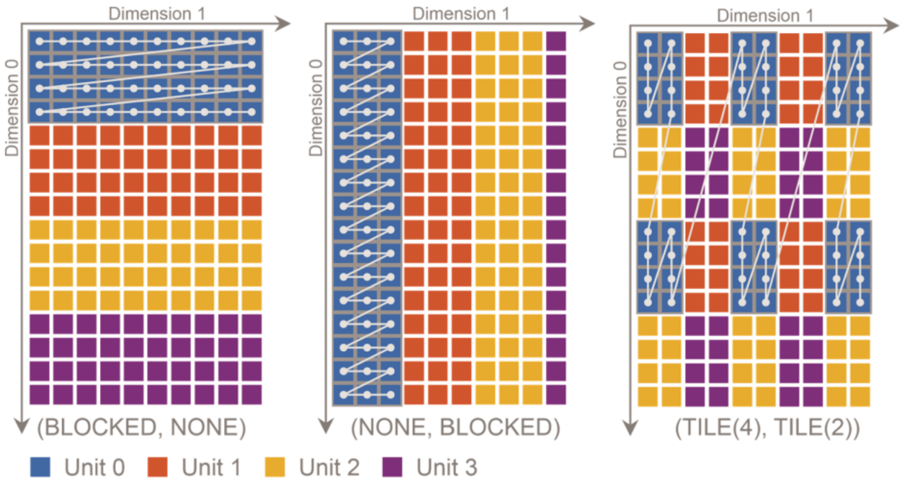
The N-Dimensional Array

- `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; // local iterator

int c = loc(i,j);
int d = *(loc.begin());
```

Multidimensional Data Distribution

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



DASH Algorithms

- Growing number of **DASH** equivalents for **STL** algorithms

```
dash::GlobIter<T> dash::fill(GlobIter<T> begin, GlobIter<T> end, val);
```

- Examples of STL algorithms ported to DASH
(which also work for multidimensional ranges)

```
– dash::copy           range[i] <- range2[i]
– dash::fill           range[i] <- val
– dash::generate       range[i] <- func()
– dash::for_each       func(range[i])
– dash::transform      range[i] = func(range2[i])
– dash::accumulate     sum(range[i]) (0<=i<=n-1)
– dash::min_element    min(range[i]) (0<=i<=n-1)
```

DASH Algorithms

- Growing number of **DASH** equivalents for **STL** algorithms

```
dash::GlobIter<T> dash::fill(GlobIter<T> begin, GlobIter<T> end, val);
```

- Examples of STL algorithms ported to DASH
  (which also work for multidimensional ranges)

```
– dash::copy          range[i] <- range2[i]
– dash::fill          range[i] <- val
– dash::generate      range[i] <- func()
– dash::for_each      func(range[i])
– dash::transform     range[i] = func(range2[i])
– dash::accumulate    sum(range[i]) (0<=i<=n-1)
– dash::min_element   min(range[i]) (0<=i<=n-1)
```

map

DASH Algorithms

- Growing number of **DASH** equivalents for **STL** algorithms

```
dash::GlobIter<T> dash::fill(GlobIter<T> begin, GlobIter<T> end, val);
```

- Examples of STL algorithms ported to DASH
  (which also work for multidimensional ranges)

  - `dash::copy` `range[i] <- range2[i]`
  - `dash::fill` `range[i] <- val`
  - `dash::generate` `range[i] <- func()`
  - `dash::for_each` `func(range[i])`
  - `dash::transform` `range[i] = func(range2[i])`
  - `dash::accumulate` `sum(range[i]) (0<=i<=n-1)`
  - `dash::min_element` `min(range[i]) (0<=i<=n-1)`

map

reduce

DASH Algorithms

- Growing number of **DASH** equivalents for **STL** algorithms

```
dash::GlobIter<T> dash::fill(GlobIter<T> begin, GlobIter<T> end, val);
```

- Examples of STL algorithms ported to DASH
  (which also work for multidimensional ranges)

  - `dash::copy` `range[i] <- range2[i]`
  - `dash::fill` `range[i] <- val`
  - `dash::generate` `range[i] <- func()`
  - `dash::for_each` `func(range[i])`
  - `dash::transform` `range[i] = func(range2[i])`
  - `dash::accumulate` `sum(range[i]) (0<=i<=n-1)`
  - `dash::min_element` `min(range[i]) (0<=i<=n-1)`

No filter!
(remove_if)

map

reduce

DASH Algorithms

- Example – Find the minimum element in a distributed array

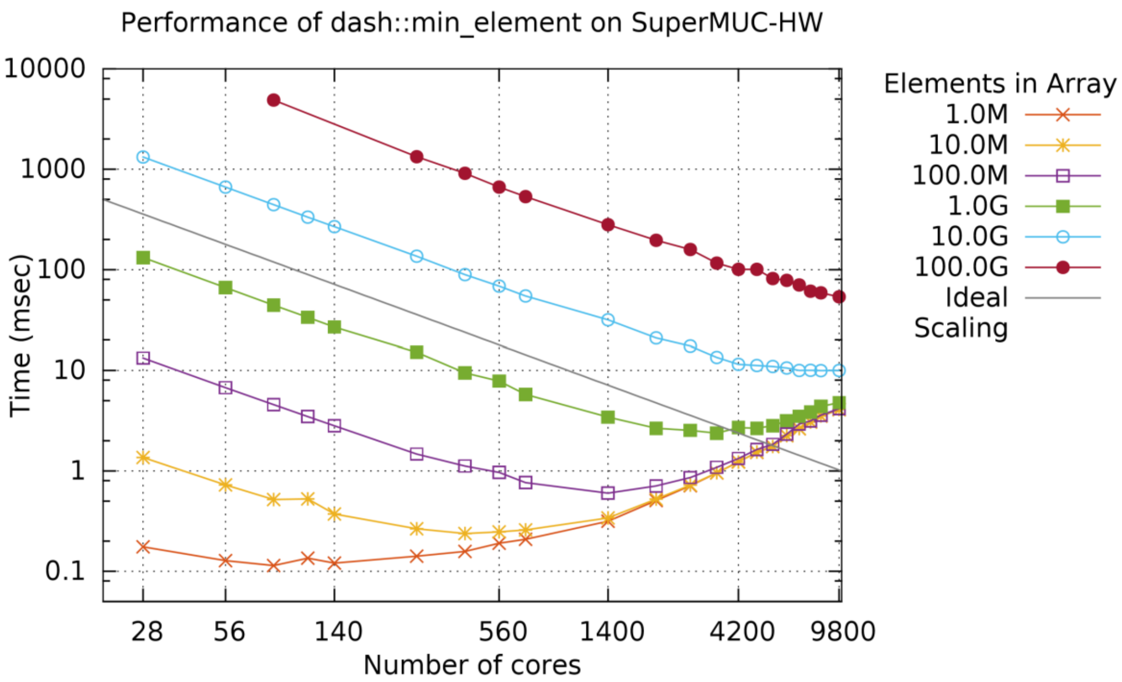
```
dash::Array<int> arr(100, dash::BLOCKED);

// ...

auto min = dash::min_element(arr.begin(), arr.end());

if (dash::myid() == 0)
    cout << "Global minimum: " << (int)*min << endl;
```

Collective call, returns global pointer to minimum element

→ reduce results of `std::min_element` of all local ranges

- Features
  - Still works when using **`CYCLIC`** or any other distribution
  - Still works when using a range other than **`[begin, end)`**

Performance of `dash::min_element()(int)`



Using DASH on Graham

Building and Installing DASH

- Home page: http://www.dash-project.org/
- Git repository: https://github.com/dash-project/dash
- Requires: `C++14` and `MPI API 3.0` or higher

```
$ cd ~/scratch
$ wget https://github.com/dash-project/dash.git
$ cd dash
$ module load gcc/7.3.0 intel/2018.3 openmpi
$ ./build.sh -DINSTALL_PREFIX=/home/$USER
$ cd build
$ make install
```

Testing and Building Programs

- Running test

```
$ cd dash
$ mpirun -n 4 ./dash-test-mpi
```

- Compiling and running DASH programs

```
$ dash-mpicxx hello.cpp -o hello_mpi
$ mpirun -n 4 ./hello_mpi
```

Dash Tutorial

https://github.com/arminms/dash-tutorial

GitHub