The good, the bad and the ugly: Experiences with developing a PGAS runtime on top of MPI-3

6th Workshop on Runtime and Operating Systems for the Many-core Era (ROME 2018)

www.dash-project.org

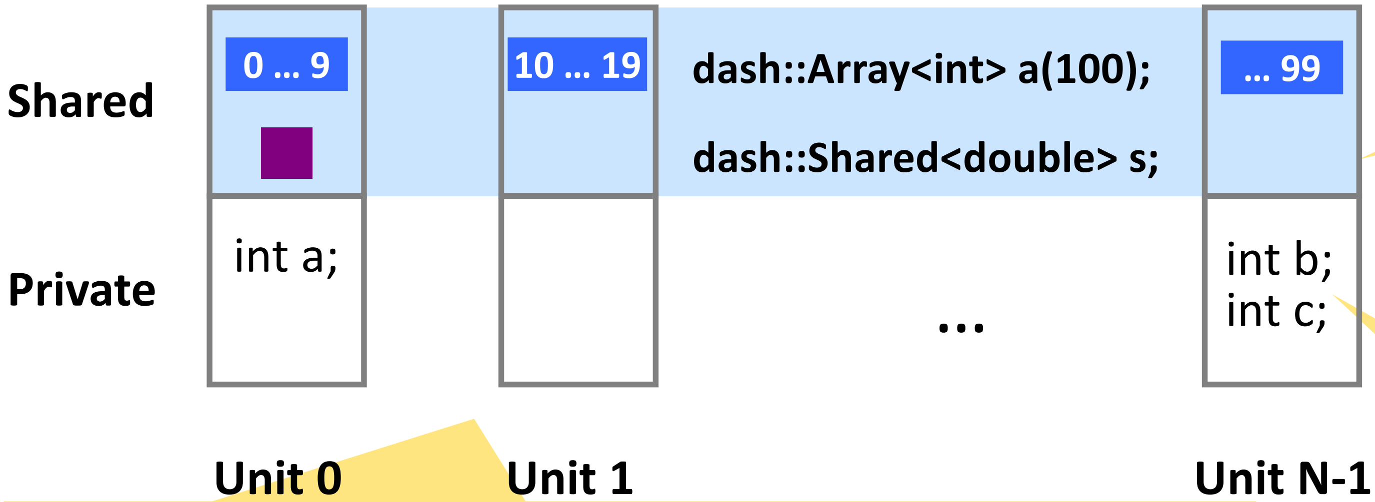
Karl Fürlinger
Ludwig-Maximilians-Universität München

(Most work presented her is by Joseph Schuchart (HLRS) and other members of the DASH team)

The Context - DASH

- DASH is a C++ template library that implements a PGAS programming model
  - Global data structures, e.g., `dash::Array<>`
  - Parallel algorithms, e.g., `dash::sort()`
  - No custom compiler needed
- Terminology

Shared data: managed by DASH in a virtual global address space



Private data: managed by regular C/C++ mechanisms

Unit: The individual participants in a DASH program, usually full OS processes. DASH follows the SPMD model

DASH Example Use Cases

- Data Structures

```
struct s {...};

dash::Array<int> arr(100);
dash::NArray<s,2> matrix(100, 200);
```

One or multi-dimensional arrays over primitive types or simple composite types (“trivially copyable”)

- Algorithms

Algorithms working in parallel on the a global range of elements

```
dash::fill(arr.begin(), arr.end(), 0);
dash::sort(matrix.begin(), matrix.end());

std::fill(arr.local.begin(),
          arr.local.end(),
          dash::myid());
```

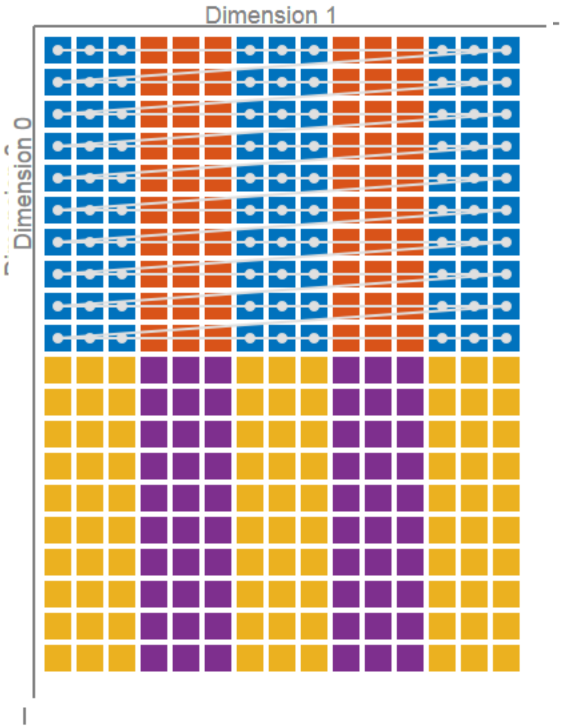
Access to locally stored data, interoperability with STL algorithms

Data Distribution and Local Data Access

- Data distribution can be specified using Patterns

Size in first and second dimension

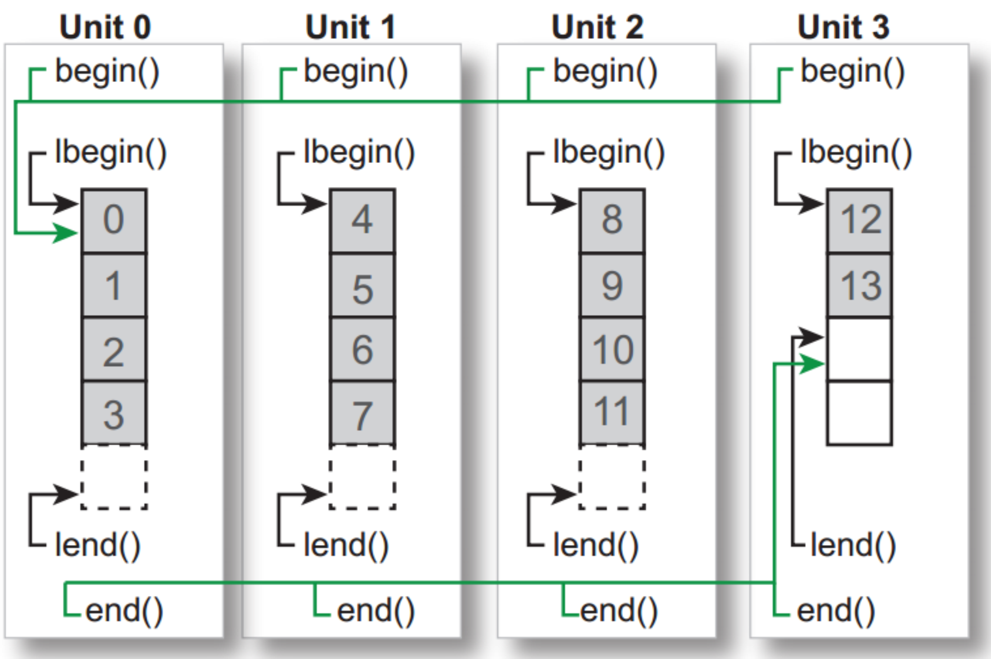
Pattern<2>(20, 15)



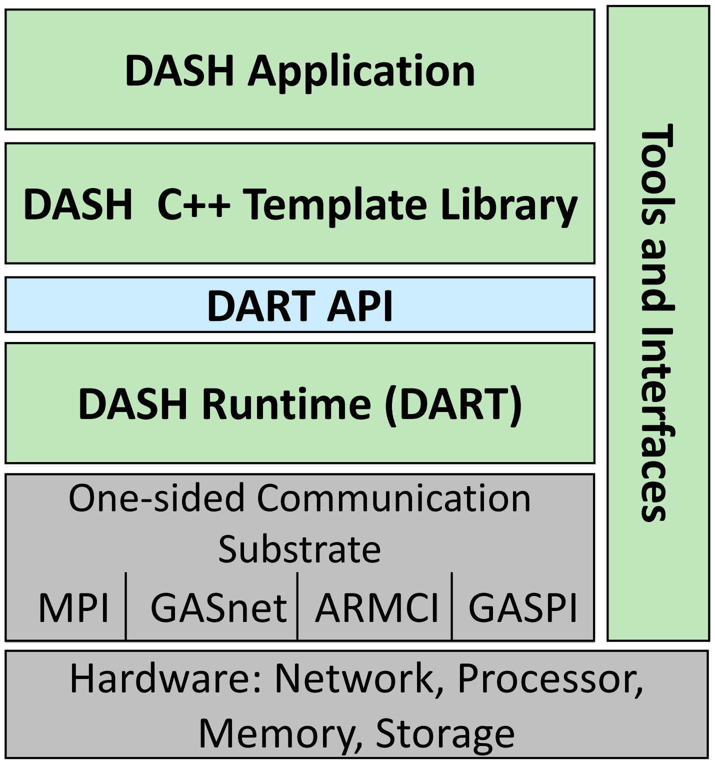
(BLOCKED, NONE)
(NONE, BLOCKCYCLIC(2))
(BLOCKED, BLOCKCYCLIC(3))

Distribution in first and second dimension

- Globalview and localview semantics



DASH — Project Structure



	Phase I (2013-2015)	Phase II (2016-2018)
LMU Munich	Project management, C++ template library	Project management, C++ tempalte library, DASH data dock
TU Dresden	Libraries and interfaces, tools support	Smart data structures, resilience
HLRS Stuttgart	DART runtime	DART runtime
KIT Karlsruhe	Application case studies	
IHR Stuttgart		Smart deployment, Application case studies

dash

www.dash-project.org

SPPEXA

DASH is one of 16 SPPEXA projects

DART

- DART is the DASH Runtime System
  - Implemented in plain C
  - Provides services to DASH, abstracts from a particular communication substrate

- DART implementations
  - DART-SHMEM, node-local shared memory, proof of concept
  - DART-CUDA, shared memory + CUDA, proof of concept
  - DART-GASPI, for evaluating GASPI

  - **DART-MPI:** Uses MPI-3 RMA, ships with DASH

    https://github.com/dash-project/dash/

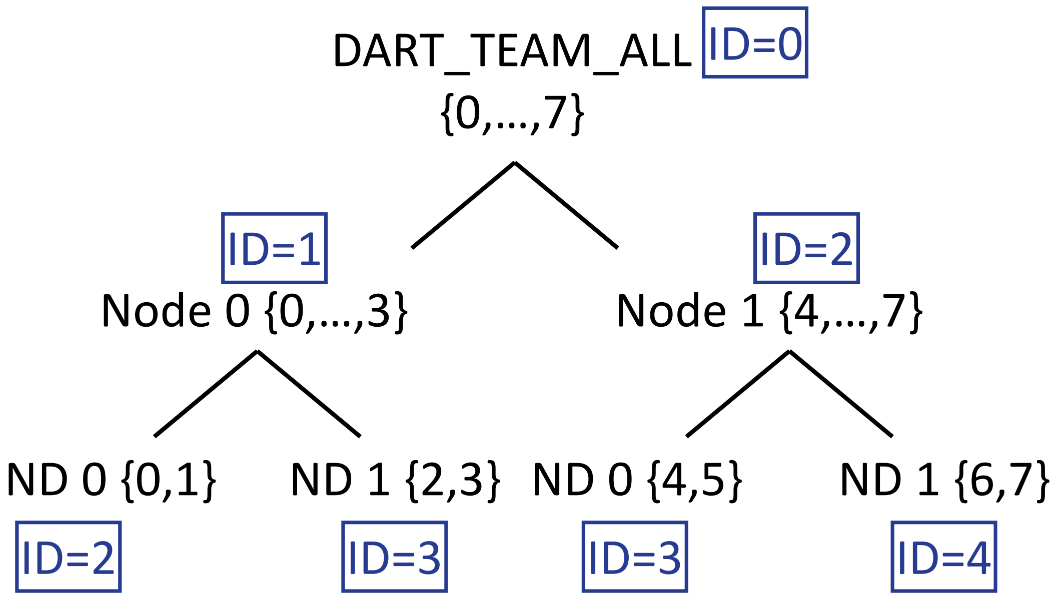
Services Provided by DART

- Memory allocation and addressing
  - Global memory abstraction, global pointers
- One-sided communication operations
  - Puts, gets, atomics
- Data synchronization
  - Data consistency guarantees
- Process groups and collectives
  - Hierarchical teams
  - Regular two-sided collectives

Process Groups

- DASH has a concept of hierarchical teams

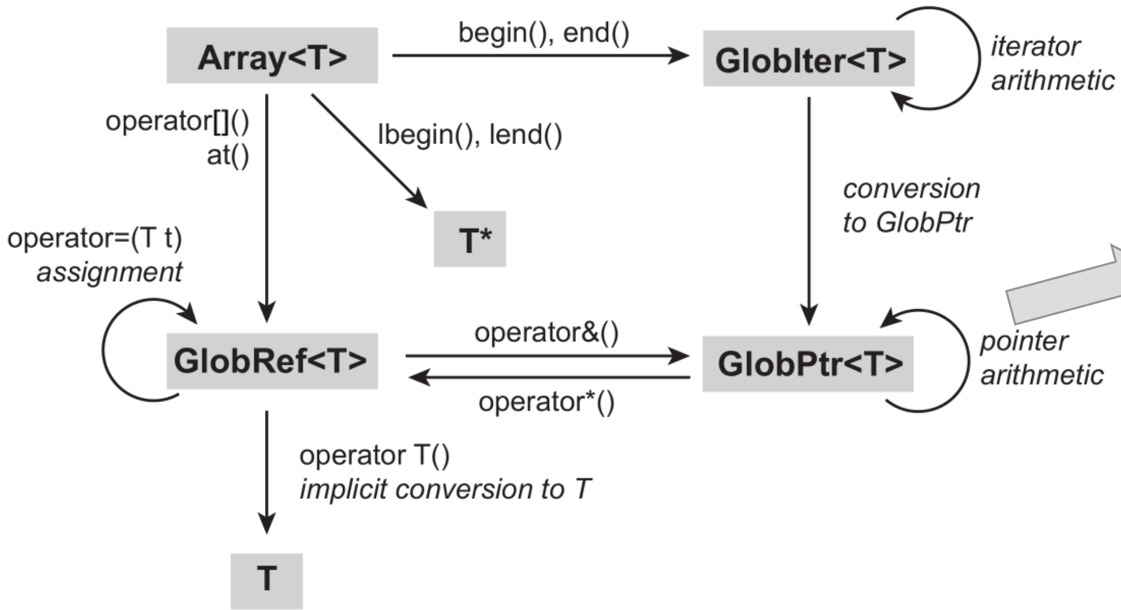
```
// get explict handle to All()
dash::Team& t0 = dash::Team::All();

// use t0 to allocate array
dash::Array<int> arr2(100, t0);

// same as the following
dash::Array<int> arr1(100);

// split team and allocate
// array over t1
auto t1 = t0.split(2)
dash::Array<int> arr3(100, t1);
```

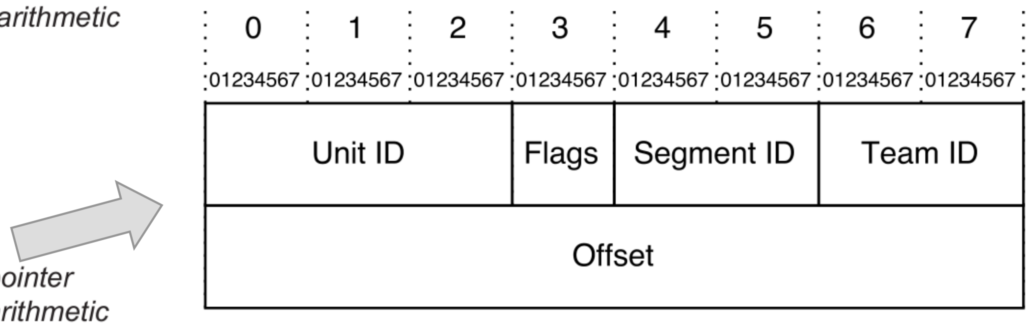


- In DART-MPI, teams map to MPI communicators
  - Splitting teams is done by using the MPI group operations

Memory Allocation and Addressing

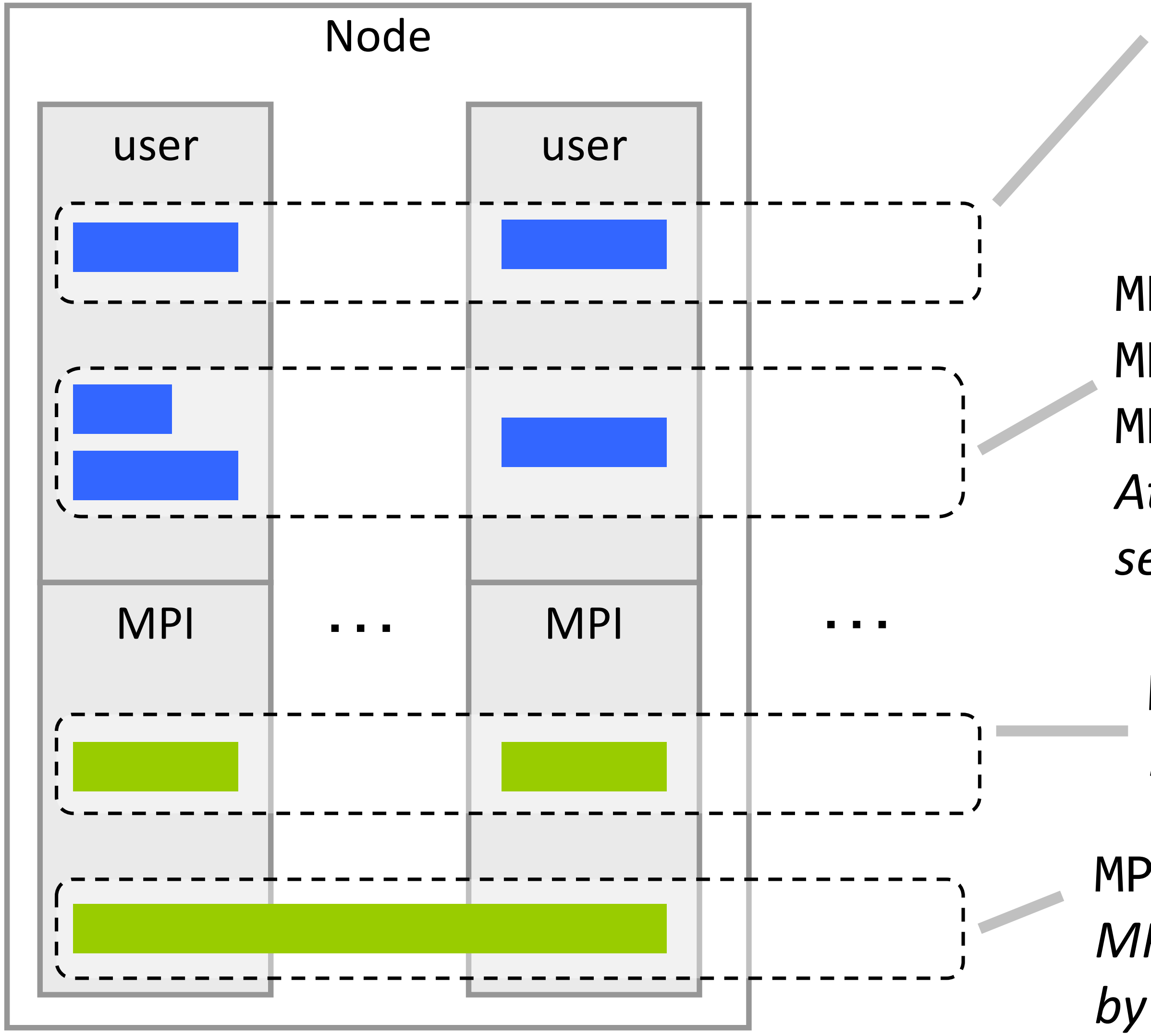
- DASH constructs a virtual global address space over multiple nodes
  - Global pointers
  - Global references
  - Global iterators





- DART global pointer
  - Segment ID corresponds to allocated MPI window

Memory Allocation Options in MPI-3 RMA



`MPI_Win_create()`
User-provided memory

`MPI_Win_create_dynamic()`
`MPI_Win_attach()`
`MPI_Win_detach()`
Attach any number of memory segments

`MPI_Win_allocate()`
MPI allocates the memory

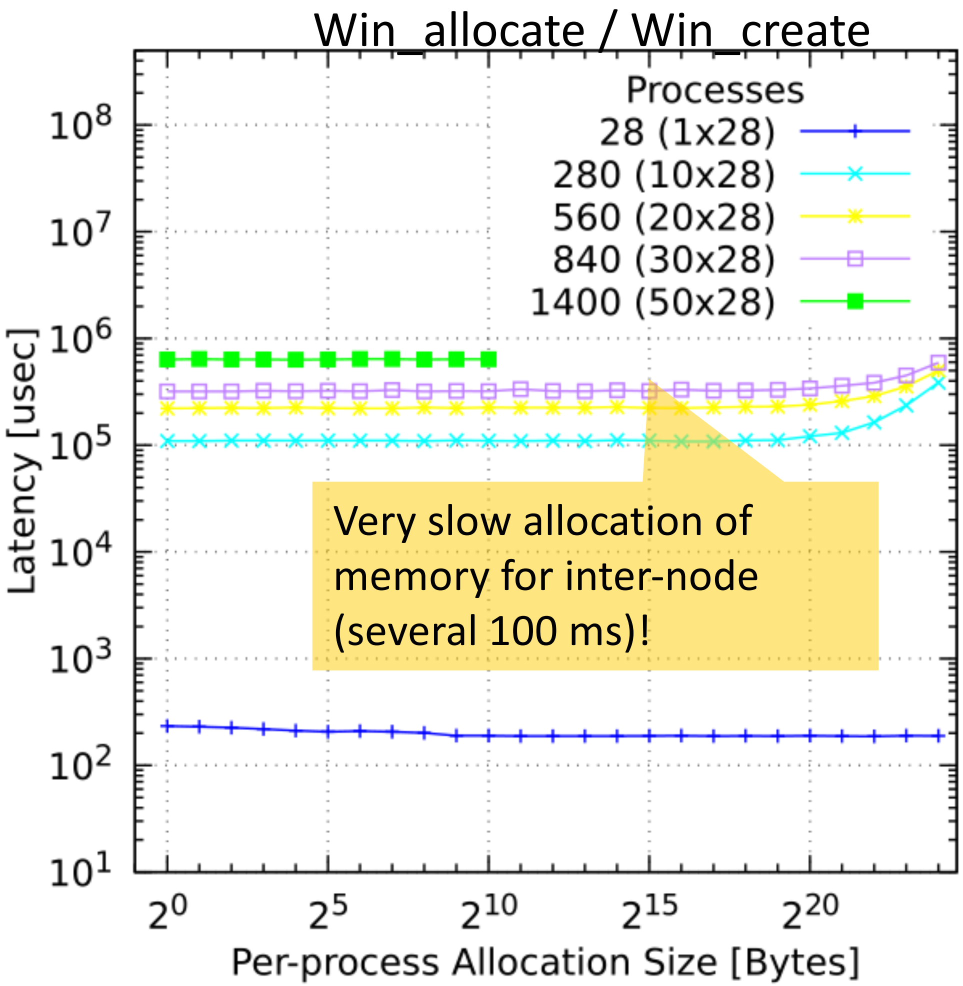
`MPI_Win_allocate_shared()`
MPI allocates memory, accessible by all ranks on a shared memory node

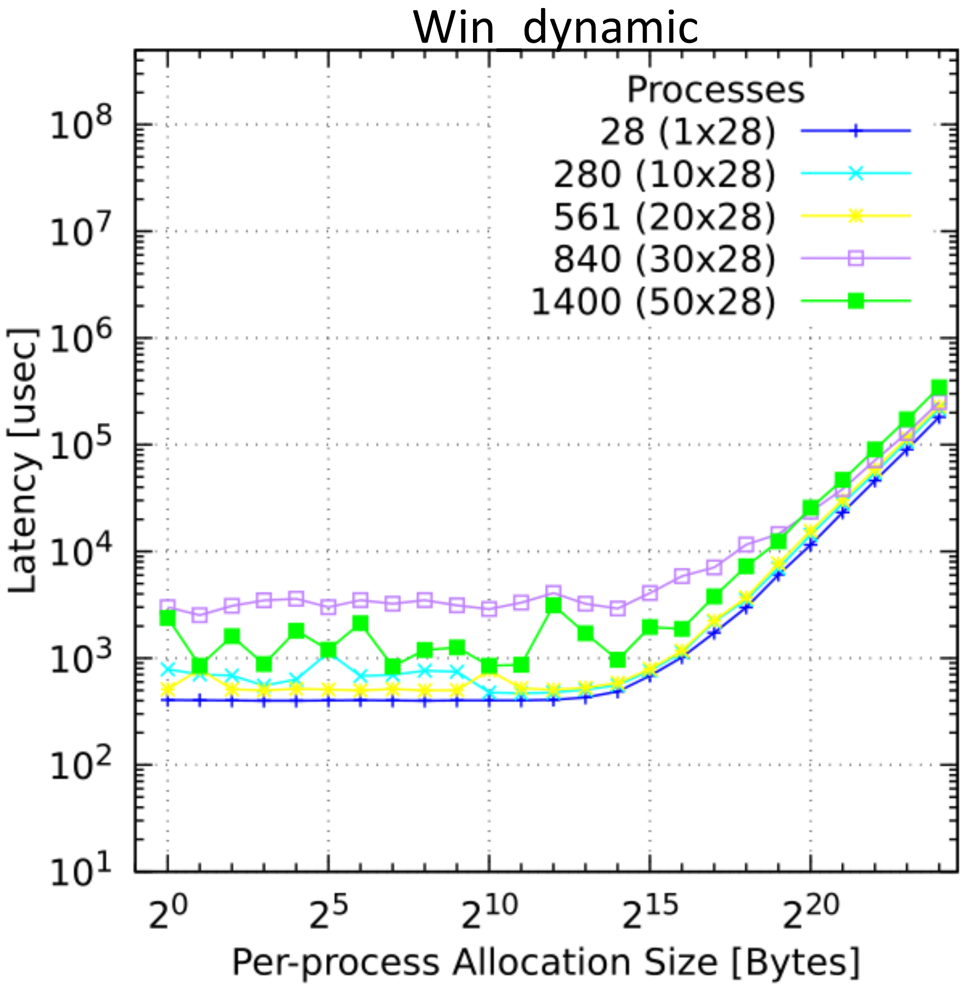
Memory Allocation Options in MPI-3 RMA

- Not immediately obvious what the best option is
- In theory:
  - MPI allocated memory can be more efficient (reg. memory)
  - Shared memory windows area a great way to optimize node-local accesses, DART can shortcut puts and gets and use regular memory access instead
- In practice
  - Allocation speed is also relevant for DASH
  - Some MPI implementations don’t support shared memory windows (E.g., IBM MPI on SuperMUC)
  - The size of shared memory windows is severely limited on some systems

Memory Allocation Latency (1)

- OpenMPI 2.0.2 on an Infiniband Cluster

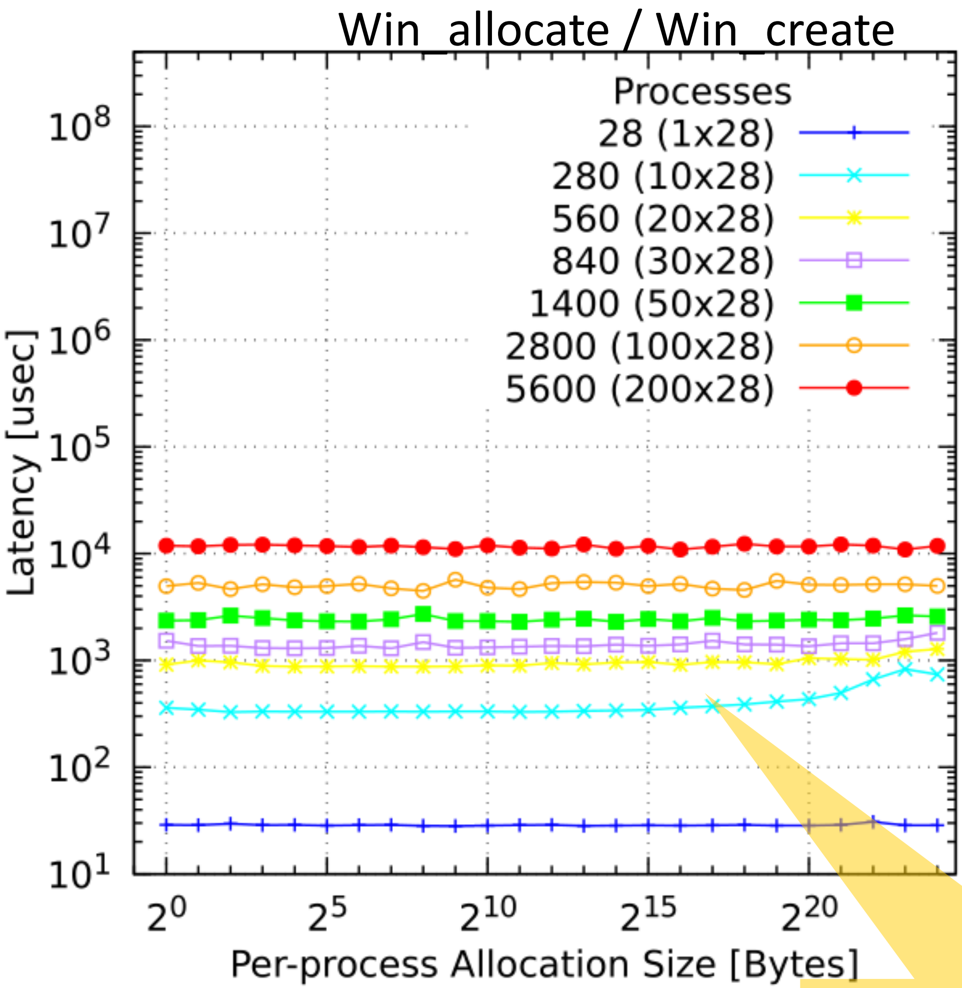


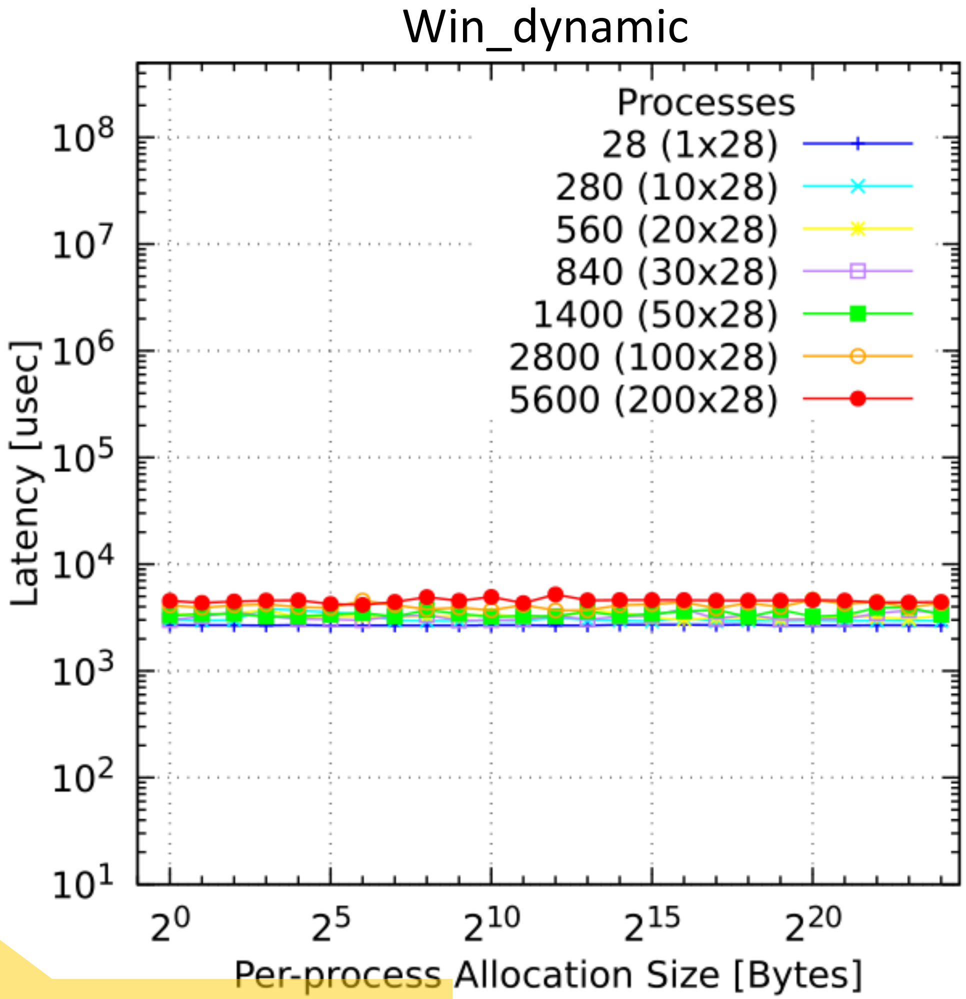


Source for all the following figures: Joseph Schuchart, Roger Kowalewski, and Karl Fürlinger. *Recent Experiences in Using MPI-3 RMA in the DASH PGAS Runtime*. In Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Workshops. Tokyo, Japan, January 2018.

Memory Allocation Latency (2)

- IBM POE 1.4 on SuperMUC

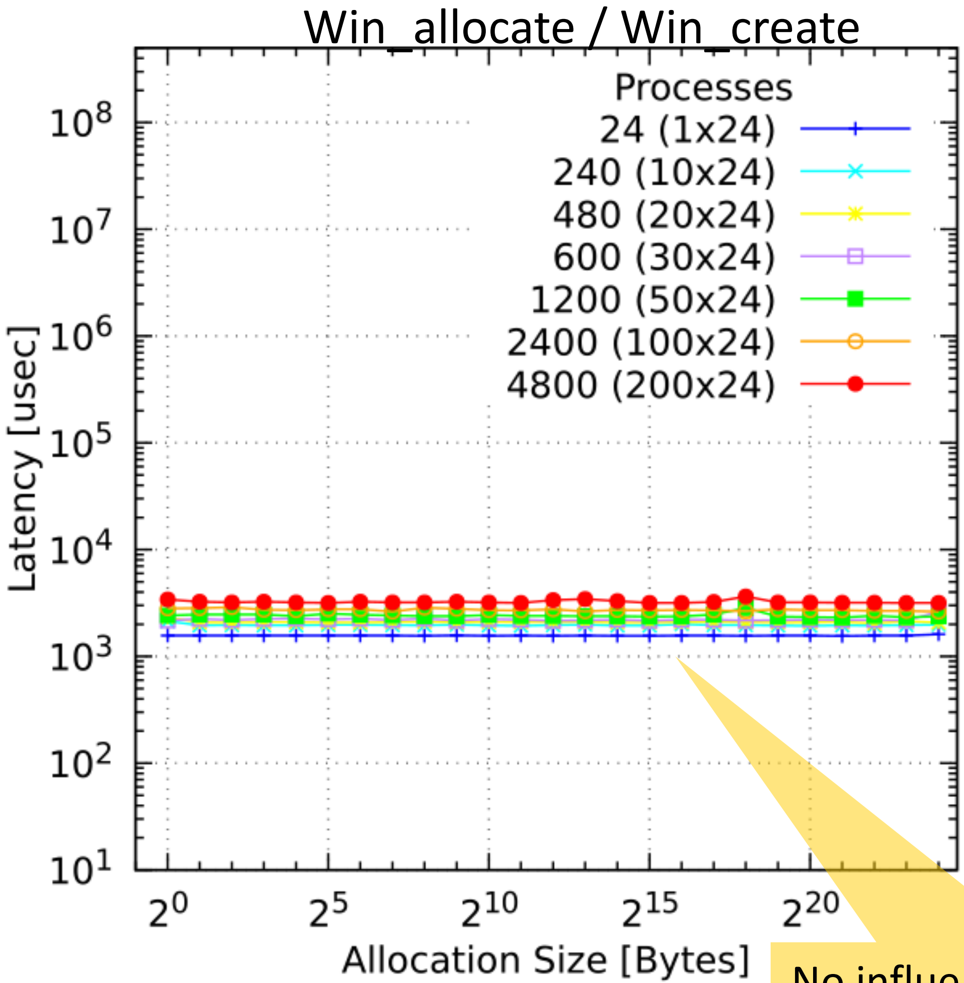


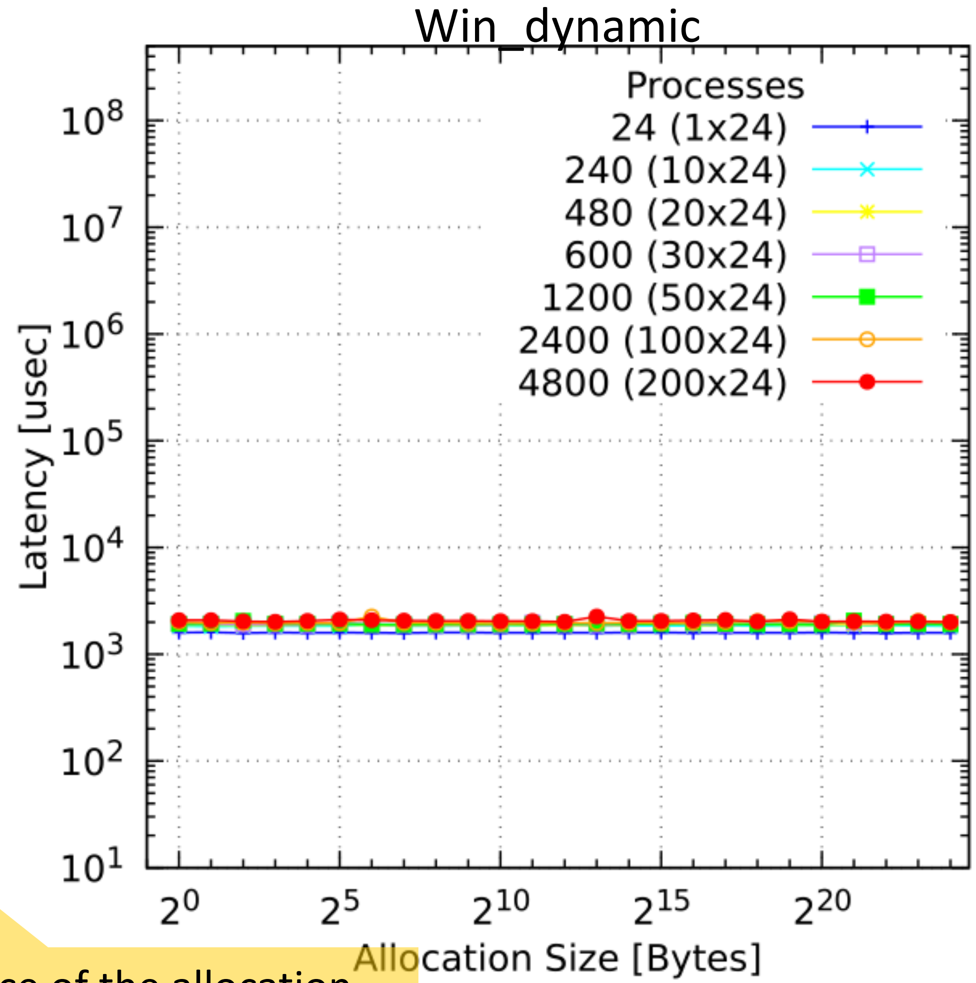


Allocation latency depends on the number of involved ranks, but not as bad as with OpenMPI

Memory Allocation Latency (3)

- Cray CCE 8.5.3 on a Cray XC40 (Hazel Hen)

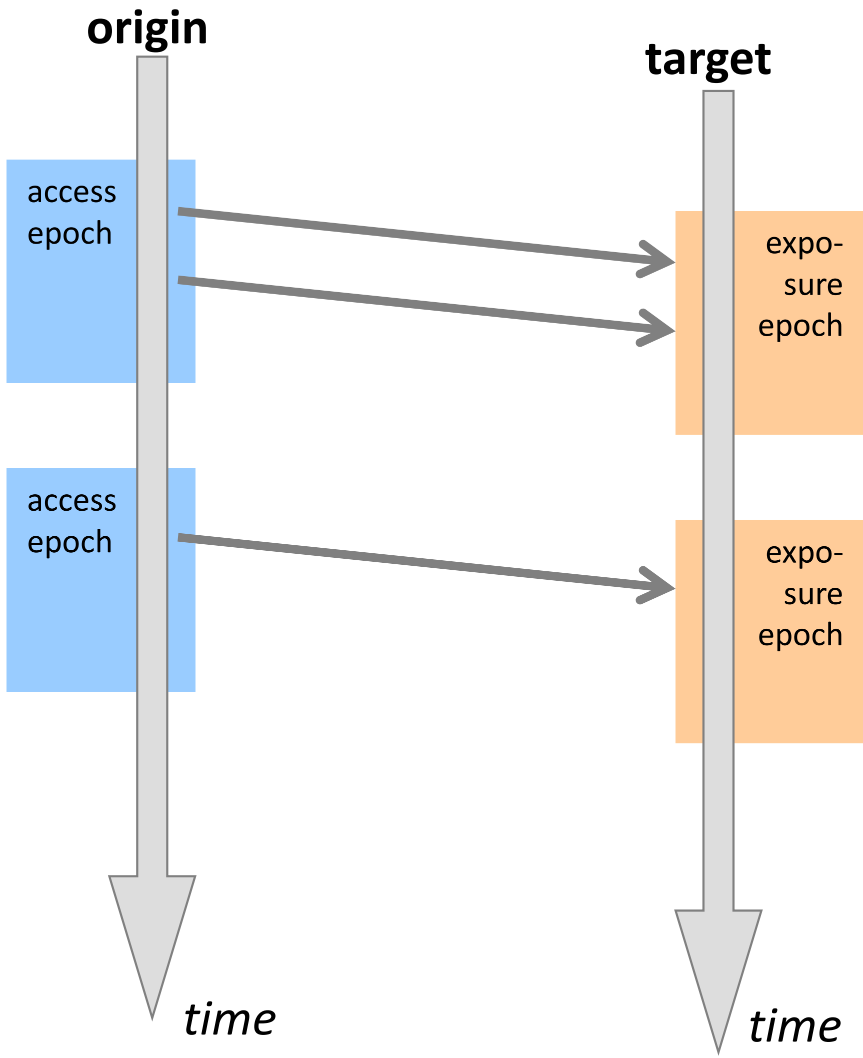




No influence of the allocation size and little influence of the number of processes.

Data Synchronization and Consistency

- Data synchronization is based on an epoch model
  - Two kinds of epochs: **access epoch** and **exposure epoch**
- Access Epoch
  - Duration of time (on the origin process) during which it may issue RMA operations (with regards to a specific target process or a group of target processes)
- Exposure Epoch
  - Duration of time (on the target process) during which it may be the target of RMA operations

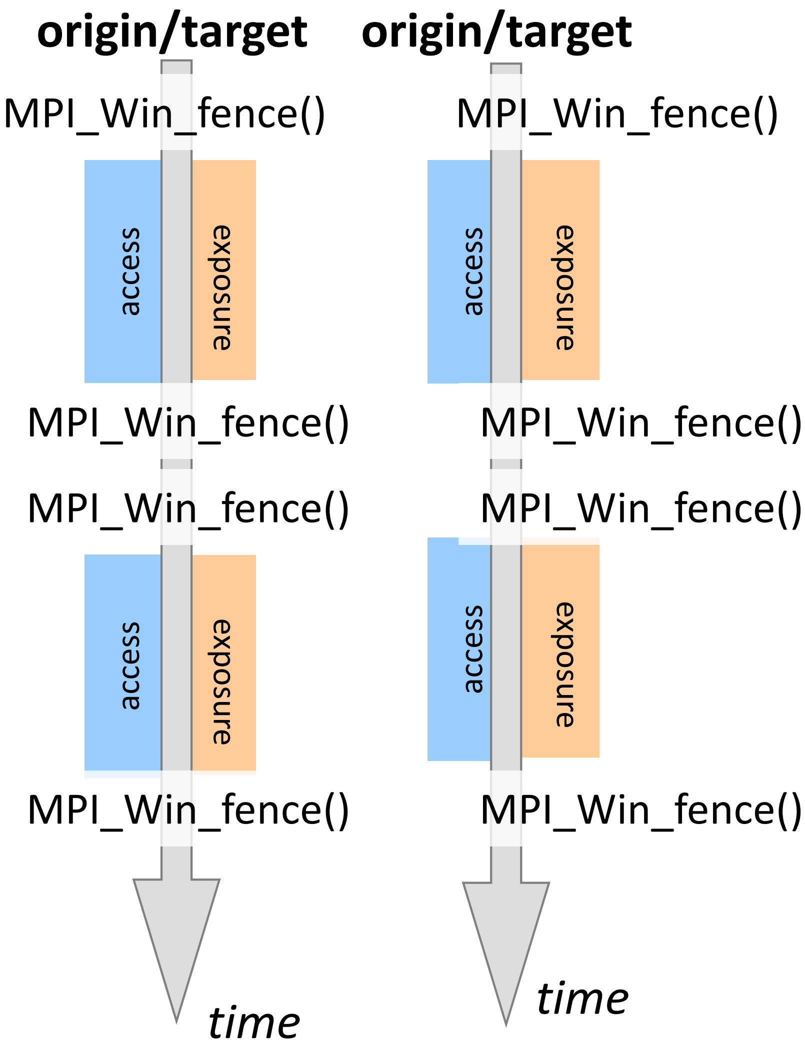


Active vs. Passive Target Synchronization

- Active target means that the target actively has to issue synchronization calls
  - Fence-based synchronization
  - General active-target synchronization, aka. PSCW: post-start-complete-wait

- Passive target means that the target does not have to actively issue synchronization calls
  - "Lock" based model

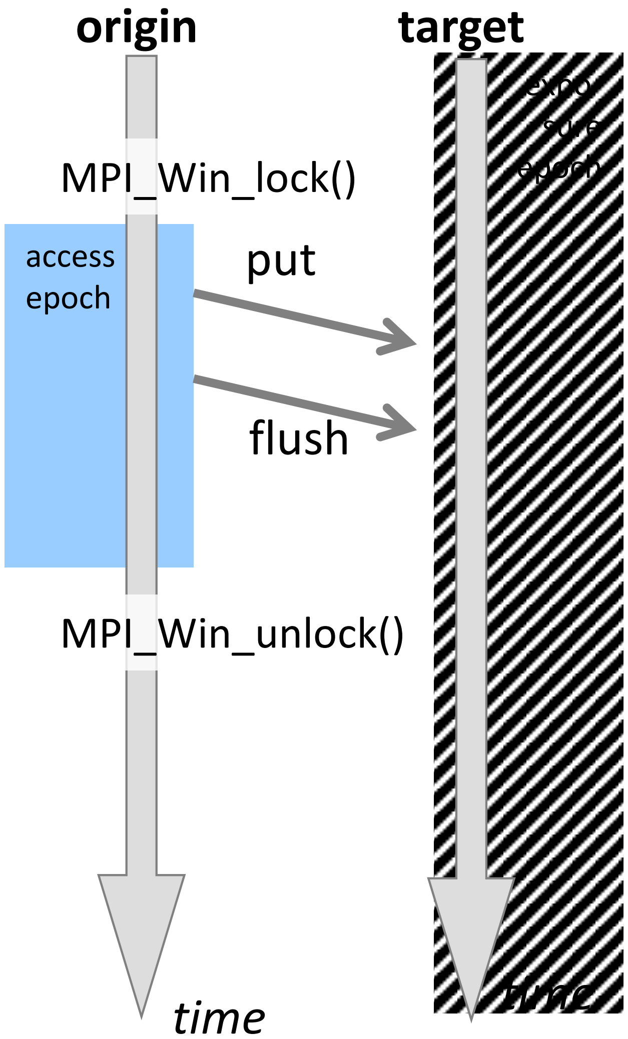
Active-Target: Fence and PSCW




```c
int MPI_Win_fence(int assert, MPI_Win win);
```

- Fence
  - Simple model, but does not fit PGAS very well
- Post/Start/Complete/Wait
  - Is more general but still not a good fit

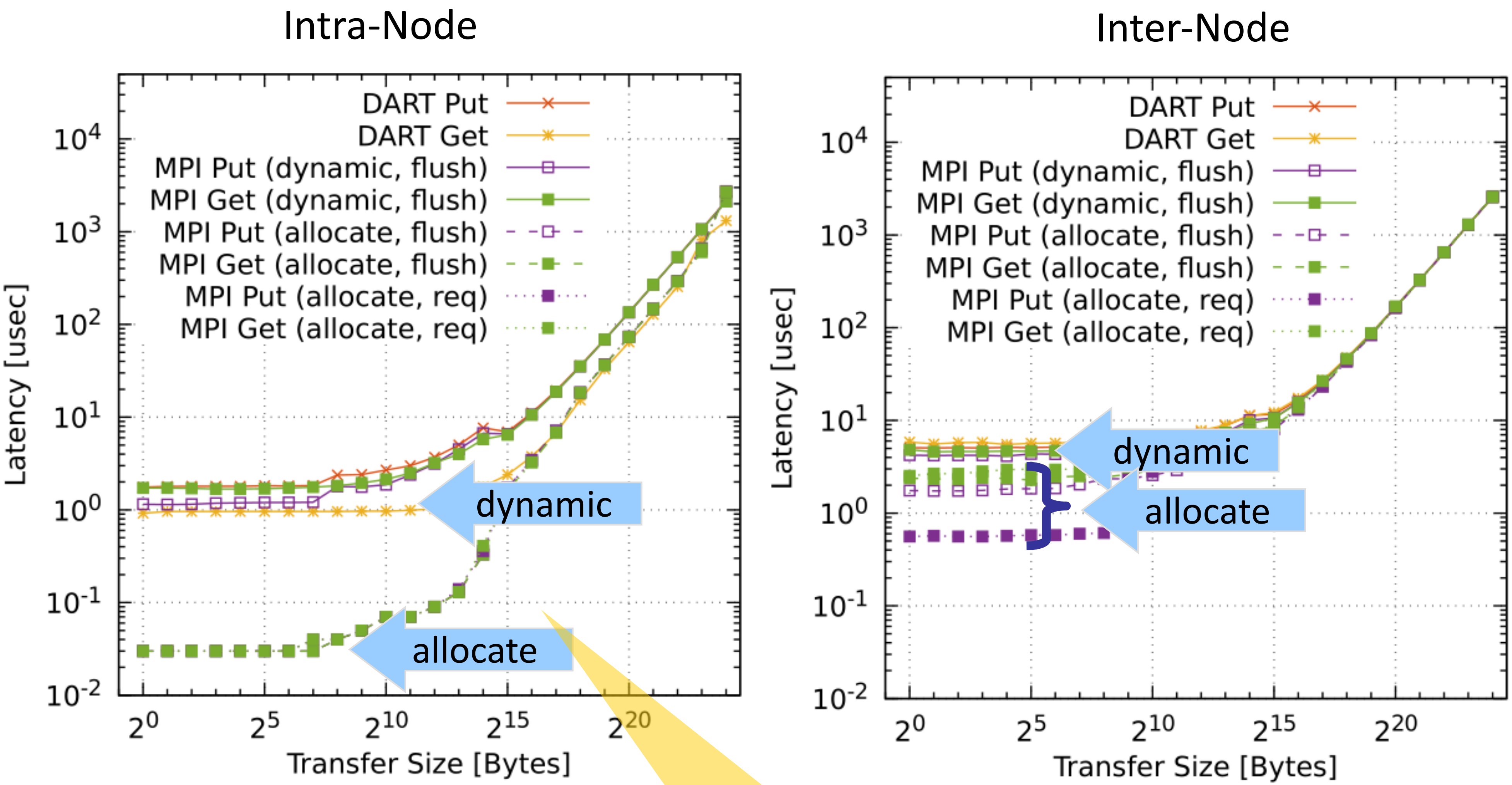
Passive-Target




```c
int MPI_Win_lock(int lock_type, int rank,
                 int assert, MPI Win win);
int MPI_Win_unlock(int rank, MPI Win win);

int MPI_Win_lock_all(int assert, MPI Win win);
int MPI_Win_unlock_all(MPI Win win);
```

- Best fit for the PGAS model, used by DART-MPI
  - One call to MPI_Win_lock_all in the beginning (after allocation)
  - One call to MPI_Win_unlock_all in the end (before deallocation)
- Flush for additional synchronization
  - MPI_Win_flush_local for **local completion**
  - MPI_Win_flush for local and **remote completion**
- Request-based operations (**MPI_Rput**, **MPI_Rget**) (only for ensuring local completion)

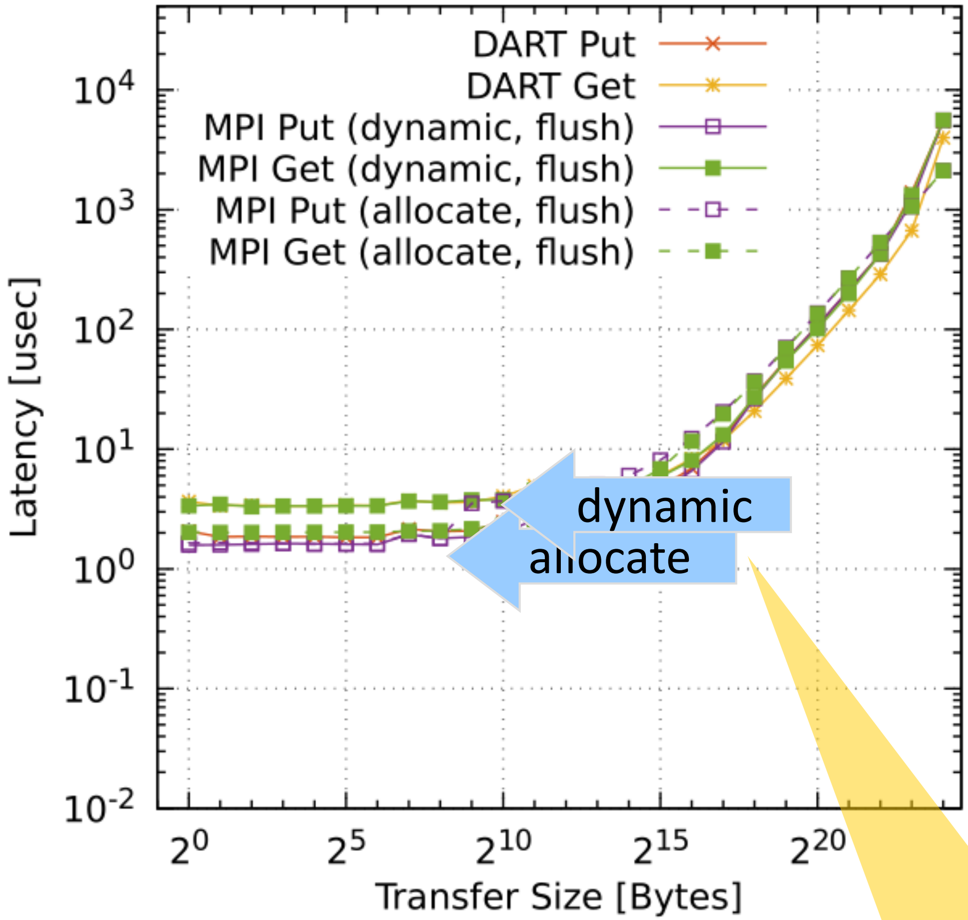
Transfer Latency: OpenMPI 2.0.2 on an Infiniband Cluster



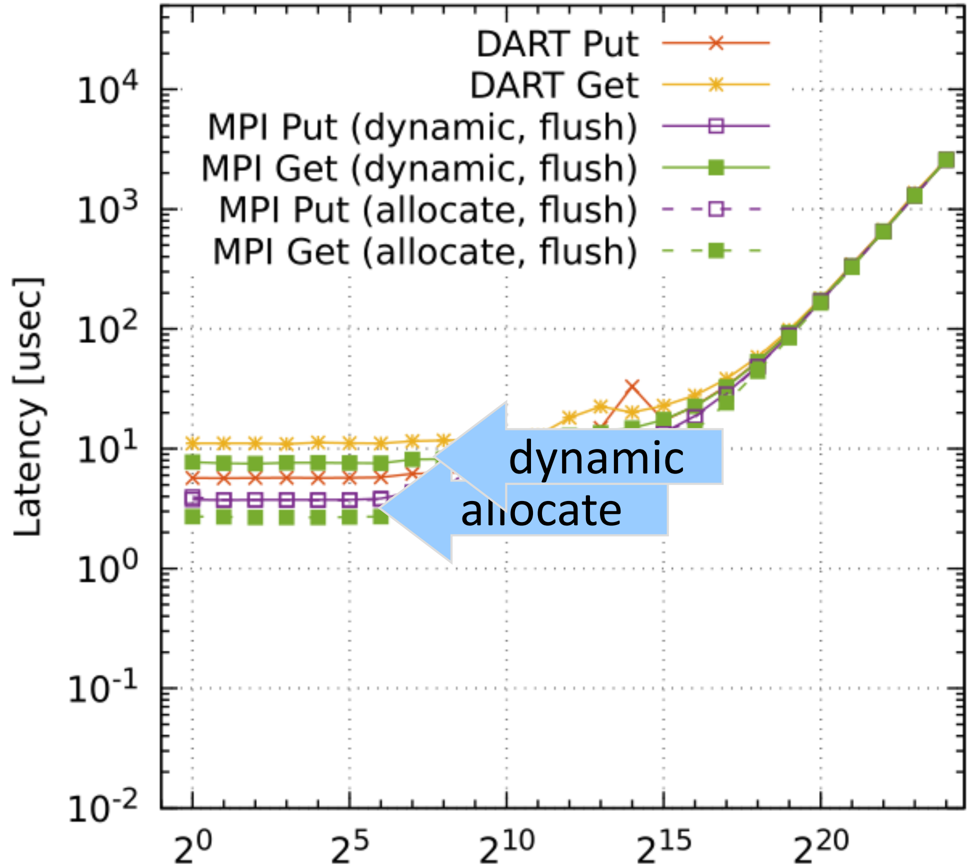
Big difference between memory allocated with Win_dynamic and Win_allocate

Transfer Latency: IBM POE 1.4 on SuperMUC

Intra-Node



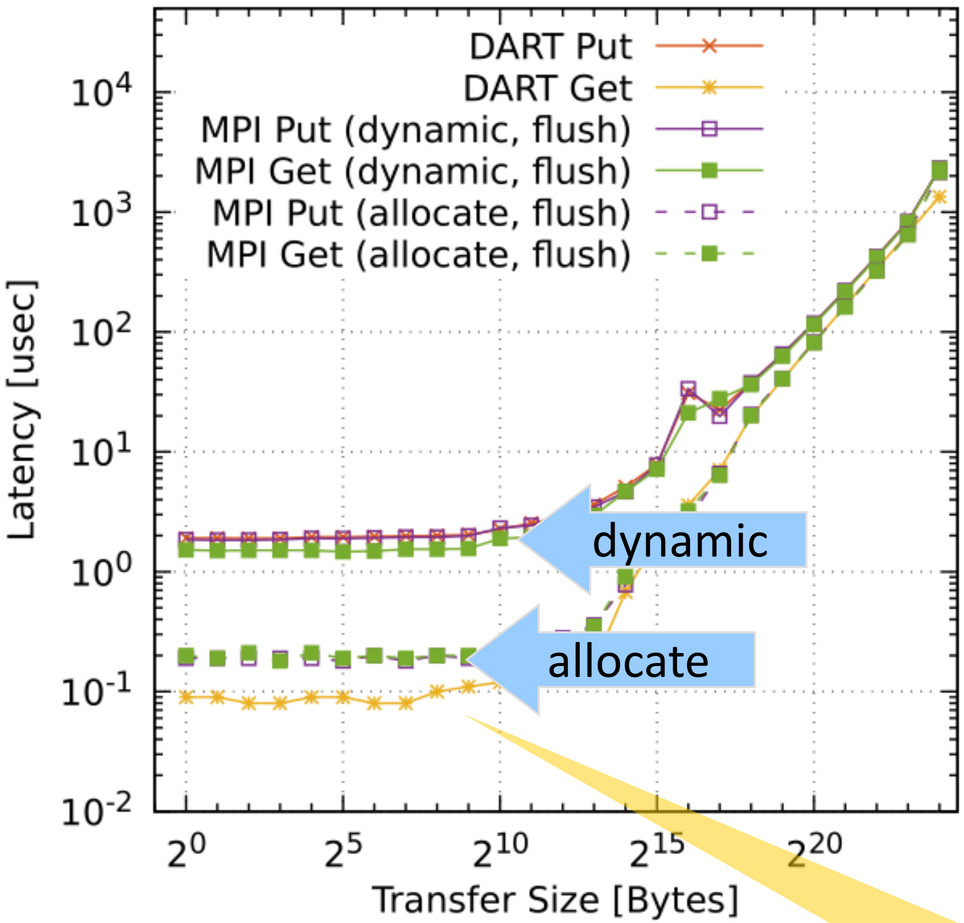
Inter-Node



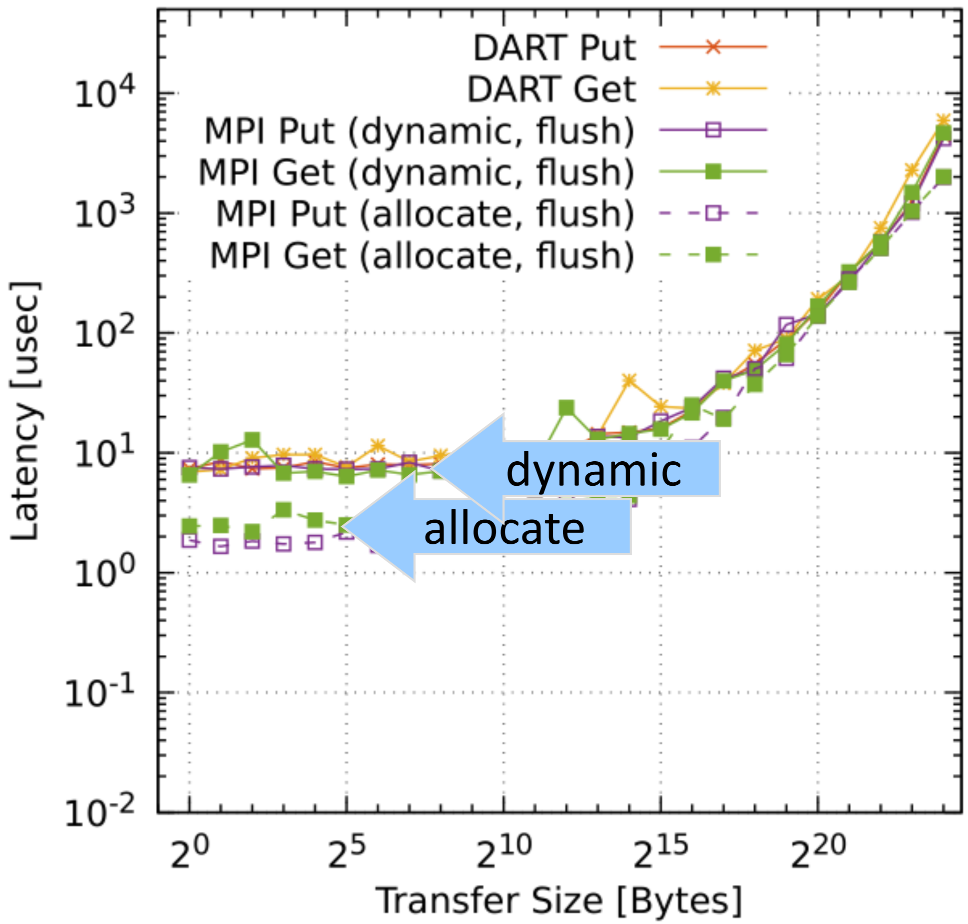
Only a small advantage of Win_allocate memory, sometimes none.

Transfer Latency: Cray CCE 8.5.3 on a Cray XC40 (Hazel Hen)

Intra-Node



Inter-Node



Significant advantages of bypassing MPI using shared memory windows.

Efficiency of Local Memory Access

```
// do some work and measure how long it takes
double do_work(int *beg, int nelem) {
  const int LCG_A = 1664525, LCG_C = 1013904223;

  int seed = 31337;
  double start, end;

  start = TIMESTAMP();
  for( int i=0; i<nelem; ++i ) {
    seed = LCG_A * seed + LCG_C;
    beg[i] = ((unsigned)seed) %100;
  }
  end = TIMESTAMP();

  return end-start;
}
```

```
dash::Array<int> arr(...)
int *mem = (int*) malloc(sizeof(int)*nelem);

double dur1 = do_work(arr.lbegin(), nelem, 1);
double dur2 = do_work(mem,          nelem, 1);
```

- Baseline (malloc): 0.012s
- Intel MPI on SuperMUC:

	D	ND
S	0.145s	0.228s
NS	0.013s	0.149s

- Workarounds have been identified…

Summary

- The good:
  - Availability on all HPC systems
  - Job launch
  - Collective operations: convenient and well-performing
  - Full featured specification (put/get/accumulate/atomics); exception: individual remote completion of puts

- The bad / ugly
  - Incomplete implementations (e.g., IBM MPI not supporting shared memory windows)
  - Significant performance differences among window allocation options between implementations – hard to find settings that are good on all platforms
  - Progress is under-specified in the specification and may need platform-specific tuning

Conclusions

- For DASH, DART-MPI will likely stay the default runtime system in the near future

- We are evaluating alternatives
  - GASPI – attractive because of fault tolerance features
  - GASnet
  - OpenSHMEM
  - …

Acknowledgements

- Funding

  DFG Deutsche Forschungsgemeinschaft

  Bundesministerium für Bildung und Forschung

- The DASH Team

  T. Fuchs (LMU), R. Kowalewski (LMU), D. Hünich (TUD), A. Knüpfer (TUD), J. Gracia (HLRS), C. Glass (HLRS), J. Schuchart (HLRS), F. Mößbauer (LMU), K. Fürlinger (LMU)

- DASH is on GitHub
  - https://github.com/dash-project/dash/
- Webpage
  - http://www.dash-project.org

Fork me on GitHub

dash