# 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

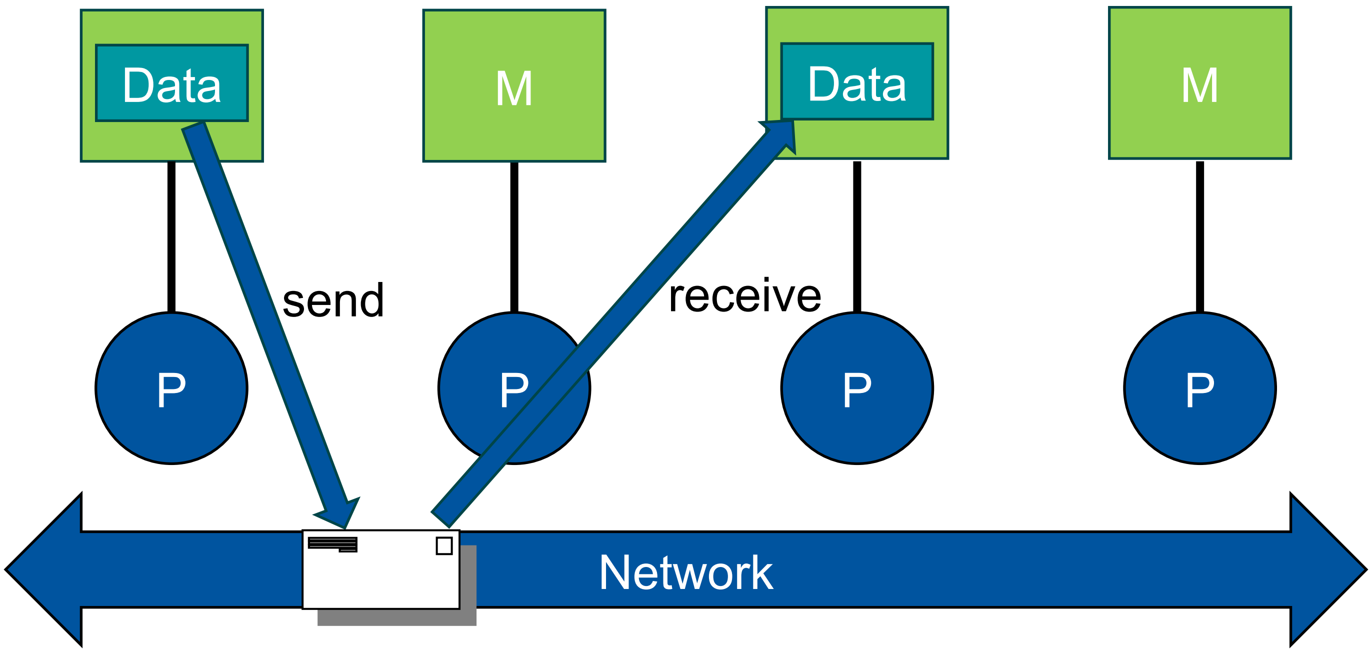
0. Organization
1. Foundations
2. Shared Memory
3. GPU Programming
4. Bulk-Synchronous Parallelism
5. Message Passing
6. **Distributed Shared Memory**
7. Parallel Algorithms
8. Parallel I/O
9. MapReduce
10. Apache Spark

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

# PGAS Foundations

# Parallel Architectures

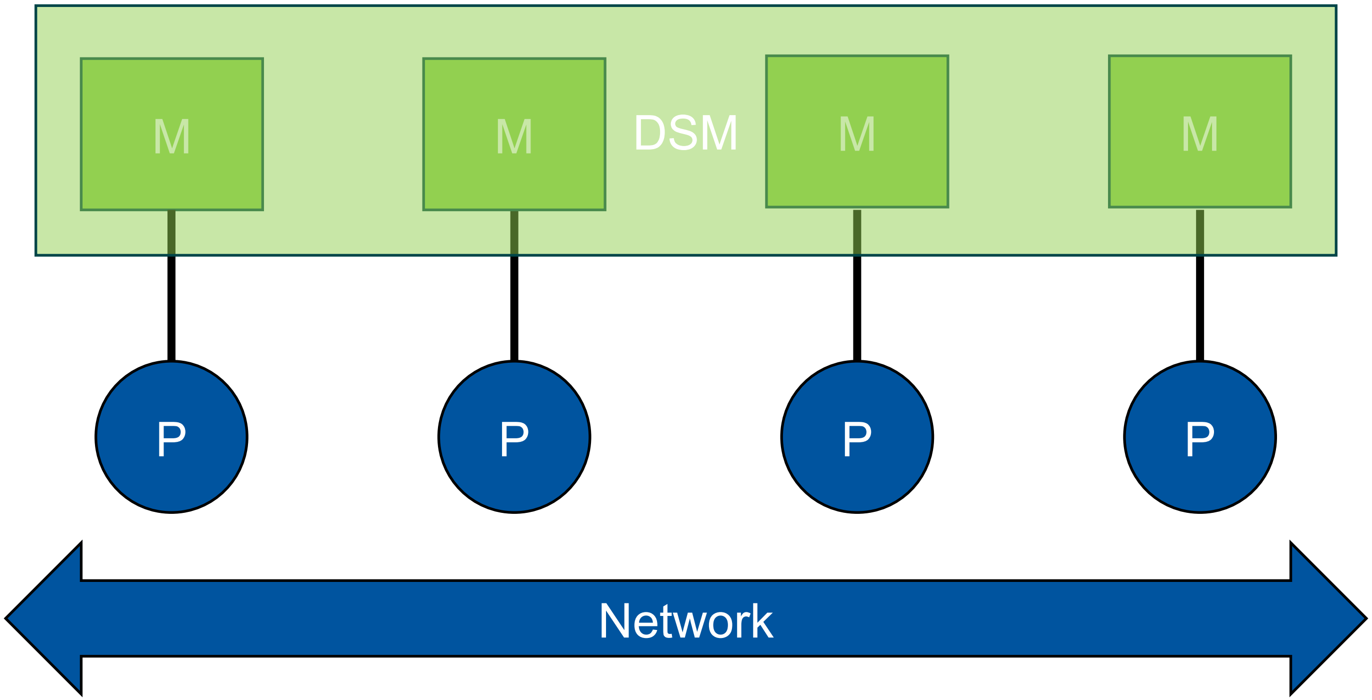
- Distributed Memory
  - Each processing element (P) has its separate main memory (M)




  - Data exchange is achieved through message passing over the network

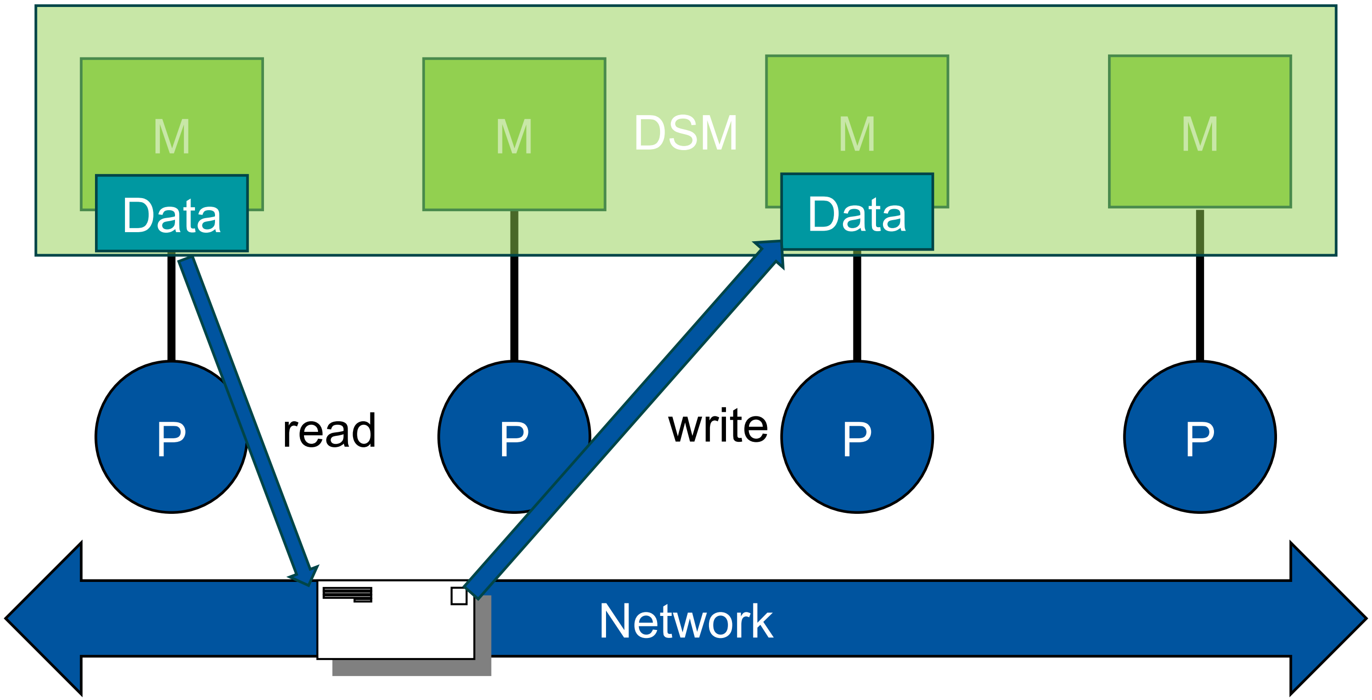
# Parallel Architectures

- Distributed Shared Memory
  - All processing elements (P) have direct access to their main memory (M)
  - Each processing element (P) has access to the shared memory (DSM)



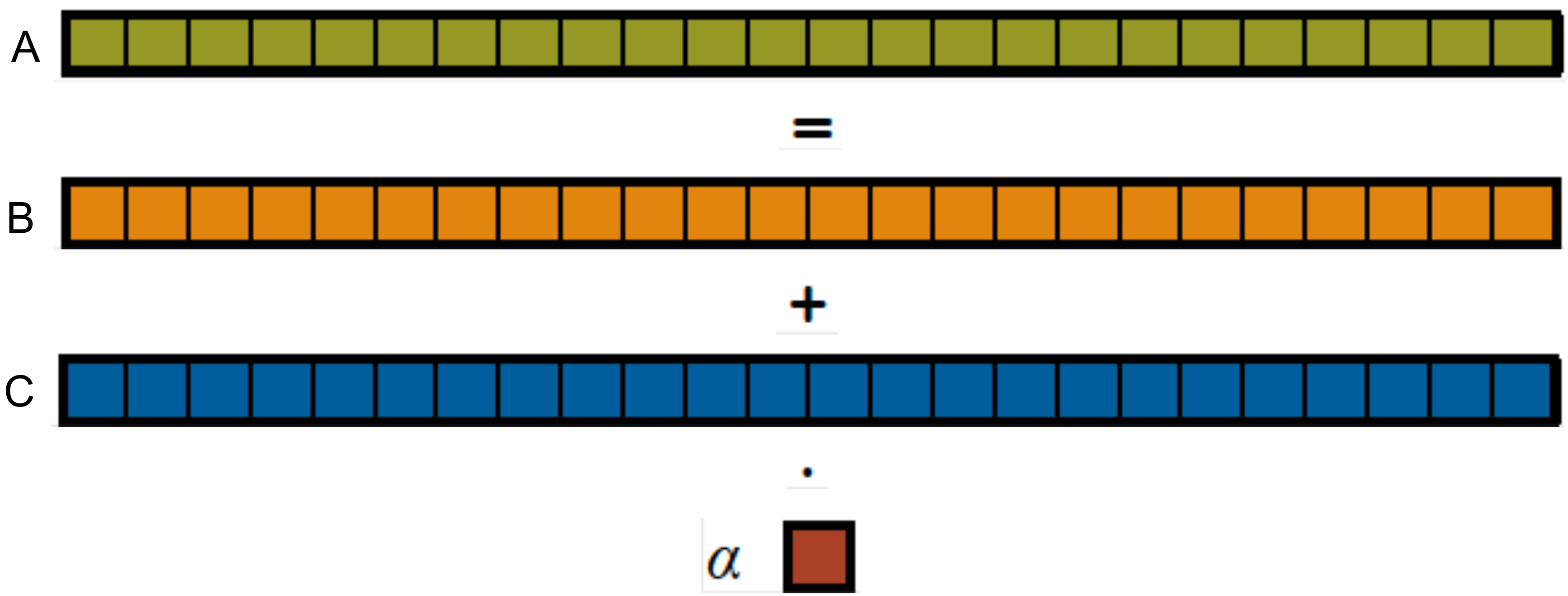
# Parallel Architectures

- Distributed Shared Memory
  - All processing elements (P) have direct access to their main memory (M)
  - Each processing element (P) has access to the shared memory (DSM)



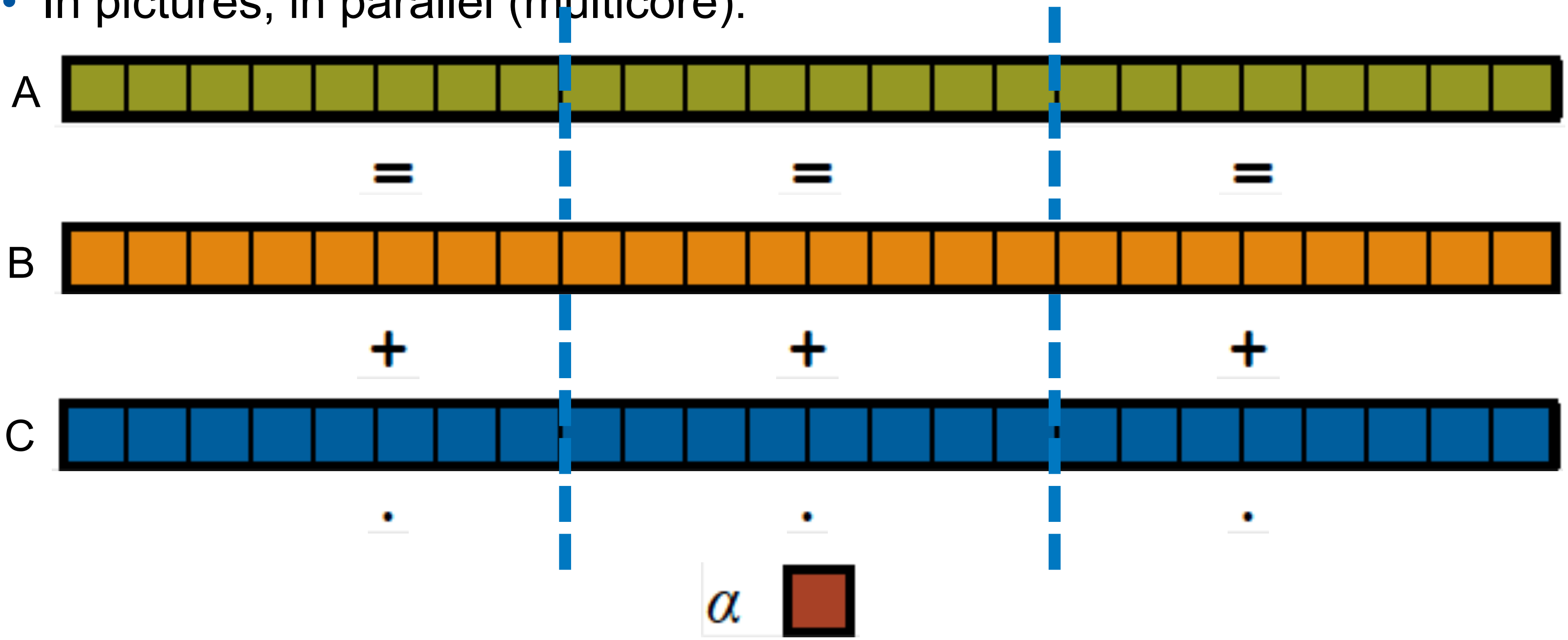
# STREAM Triad

- Given: *m*-element vectors *A*, *B*, *C*
- Compute: $\forall_{i \in 1 \ldots m}$: $\boldsymbol{A_i = B_i + \ \alpha * C_i}$
- In pictures:



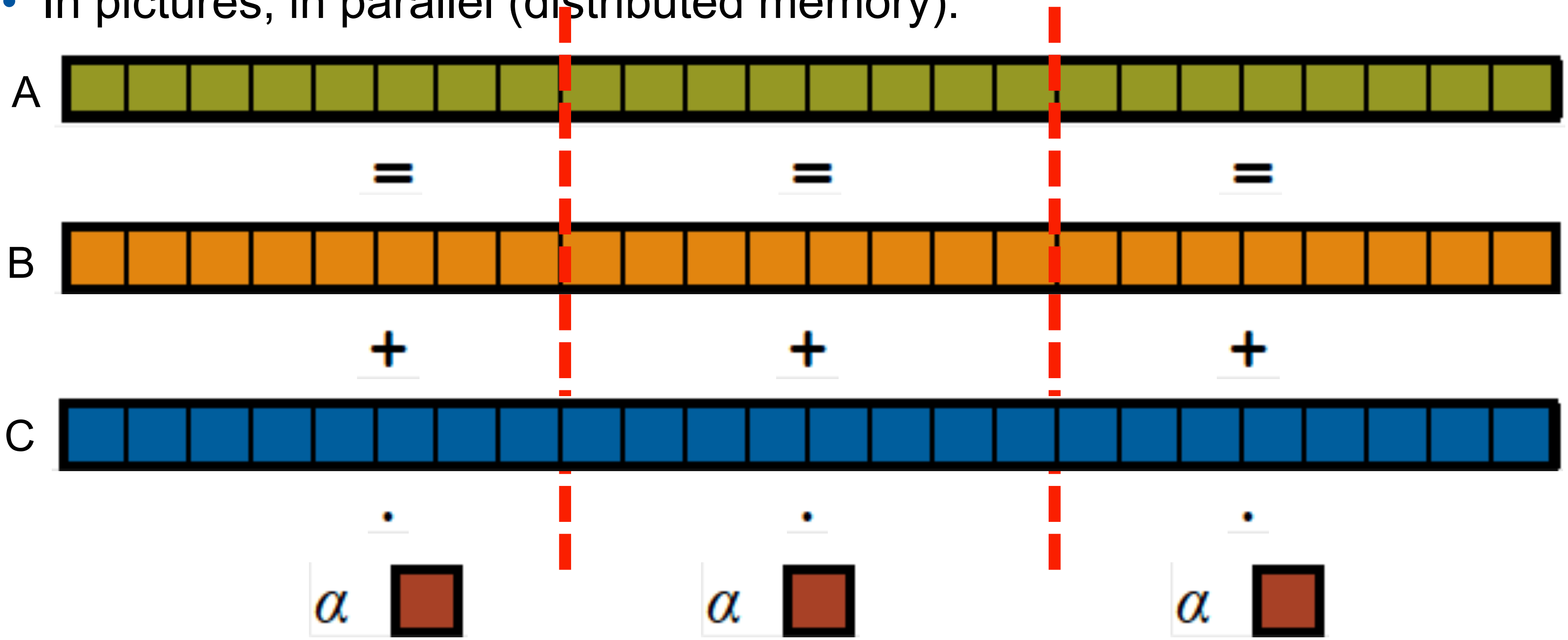
# STREAM Triad

- Given: $m$-element vectors $A$, $B$, $C$
- Compute: $\forall_{i \in 1 \ldots m}$: $\boldsymbol{A_i = B_i + \alpha * C_i}$
- In pictures, in parallel (multicore):



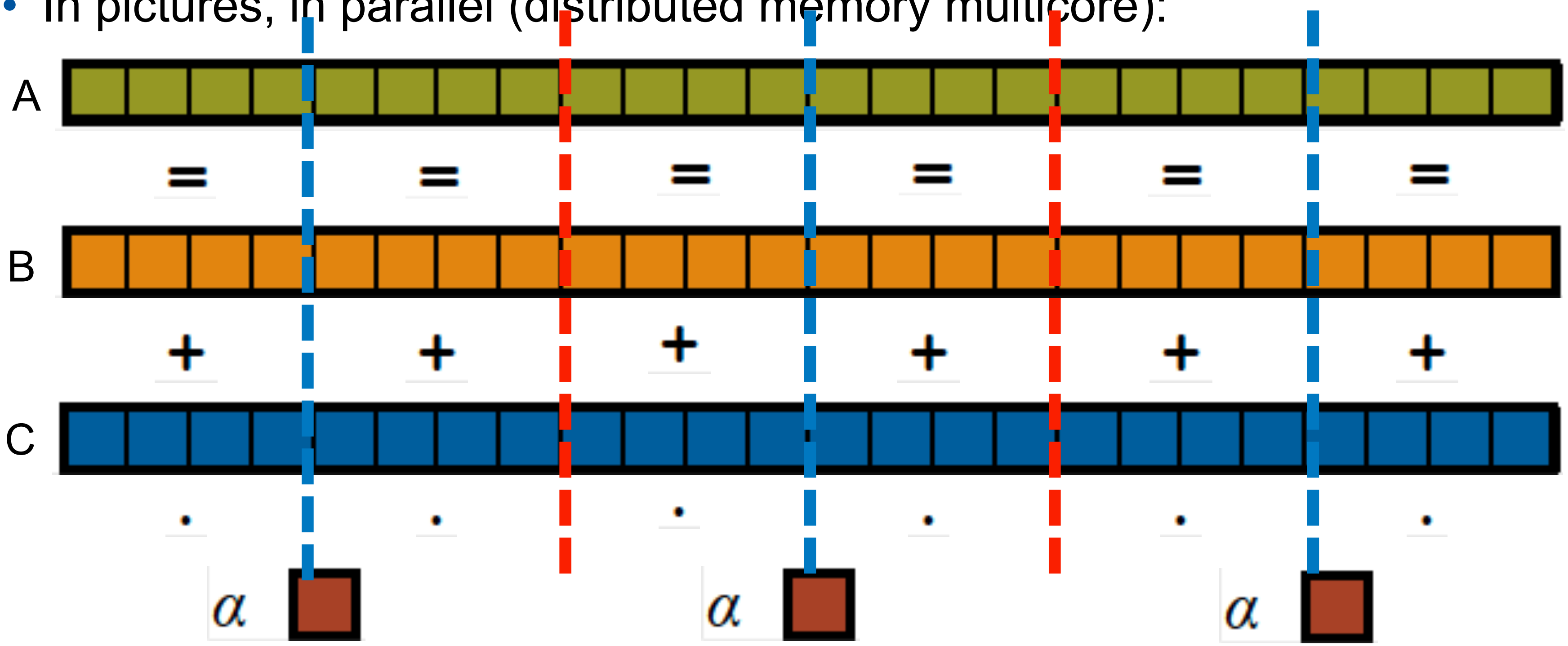
# STREAM Triad

- Given: *m*-element vectors *A*, *B*, *C*
- Compute: $\forall_{i \in 1 \ldots m}$: $\boldsymbol{A_i = B_i + \alpha * C_i}$
- In pictures, in parallel (distributed memory):



# STREAM Triad

- Given: *m*-element vectors *A*, *B*, *C*
- Compute: $\forall_{i \in 1 \ldots m}$: $\boldsymbol{A_i = B_i + \alpha * C_i}$
- In pictures, in parallel (distributed memory multicore):



# HPC STREAM Triad: MPI + OpenMP

```c
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

  rv = HPCC_Stream( params, 0 == myRank);
  MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM,
    0, comm );

  return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
  register int j;
  double  scalar;

  VectorSize = HPCC_LocalVectorSize( params, 3,
    sizeof(double), 0 );

  a = HPCC_XMALLOC( double, VectorSize );
  b = HPCC_XMALLOC( double, VectorSize );
  c = HPCC_XMALLOC( double, VectorSize );
  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
      fprintf( outFile, "Failed to allocate memory (%d).
    \n", VectorSize );
      fclose( outFile );
    }
    return 1;
  }

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 0.0;
  }

  scalar = 3.0;

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];

  HPCC_free(c);
  HPCC_free(b);
  HPCC_free(a);
```

# Parallel Programming Models

- Shared Memory, e.g., Threading
  - Support dynamic, fine-grain parallelism
  - Considered simpler, more like traditional programming
    - If you want to access something, simply name it

- But:
  - Limited scalability
  - Bugs can be subtle, difficult to track down

# Parallel Programming Models

- Distributed Memory, e.g., Message-Passing with MPI
  - More constrained model, can only access local data
  - Runs on all large-scale parallel platforms
    - And often can achieve near-optimal performance
  - Can serve as a strong foundation for high-level models

- But:
  - Communication must be used to get copies of remote data
    - Reveals (too) much about how to transfer, not what to transfer
  - Couples data transfer and synchronization
  - Has classes of bugs of its own

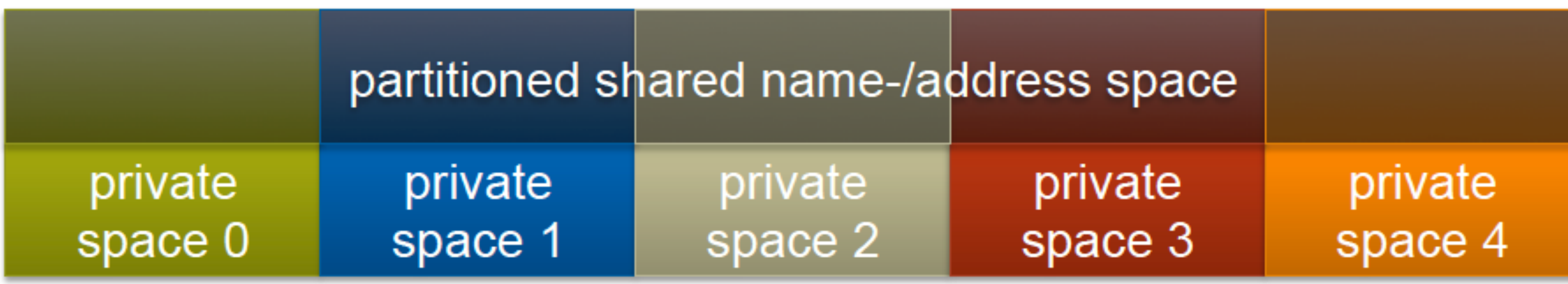
# Parallel Programming Models

- Hybrid, e.g., MPI + Threading
  - Popular in the context of HPC (Threading: OpenMP) for highest performance
  - Division of labor: each handles what it does best
  - Overheads are amortized across processor cores

- But:
  - Requires multiple notations to express single logical parallel algorithm
  - Distinct semantics of MPI and Threading / OpenMP
  - May hold surprises in terms of unexpected side effects…

# Parallel Programming Models

- Partitioned Global Address Space
  - (or: partitioned global namespace)
  - Can come as a language (extension) or API
  - Support a shared namespace on distributed memory …
    - Permit any parallel task to access any lexically visible variable
  - … with a sense of ownership
    - Variables have well-defined location, local accesses are cheaper than remote ones



# Comparison

| Name | Memory Model | Programming Model | Execution Model | Data Structures | Communica-tion |
|---|---|---|---|---|---|
| MPI | Distributed Memory | SPMD | Cooperating Executables | Manually fragmented | APIs |
| OpenMP | Shared Memory | Global-view parallelism | Threading | Shared memory arrays | - |
| BSP | Distributed Memory | SPMD | SPMD | Distributed variables, Co-arrays | Explicit and Implicit |
| PGAS | Partitioned Global Address Space | SPMD | SPMD | Co-arrays | Implicit |

- In this lecture, we present the DASH model
  - Asynchronous Partitioned Global Address Space
  - Implementation developed in the context of DFG SPPEXA prog