Optimizing Data Re-allocation Via Communication Aggregation in Chapel

Alberto Sanz, ***Rafael Asenjo***, Juan López, Rafael Larrosa, Angeles Navarro,
Vassily Litvinov, Sung-Eun Choi and Bradford L. Chamberlain

24th International Symposium on Computer Architecture and High Performance Computing
SBAC-PAD'2012
October 24-26, 2012
New York City, USA
Columbia University

Introduction

- Parallel frameworks
  - Distributed memory
    - Message passing: MPI, SHMEM, GASNet, …
  - Shared memory
    - Pthreads, OpenMP
    - Task frameworks:
      - Intel TBB, Cilk, Intel CnC, MS TPL, Java Concurrency
- Parallel Languages
  - Partitioned Global Address Space (PGAS)
    - UPC, Co-array Fortran (CAF), Titanium (Parallel Java)
  - High Performance Computing Systems (HPCS)
    - Chapel (Cray), X10 (IBM), Fortress (Sun/Oracle)
- Heterogeneous: CUDA, OpenCL, OpenACC

Chapel Motivation

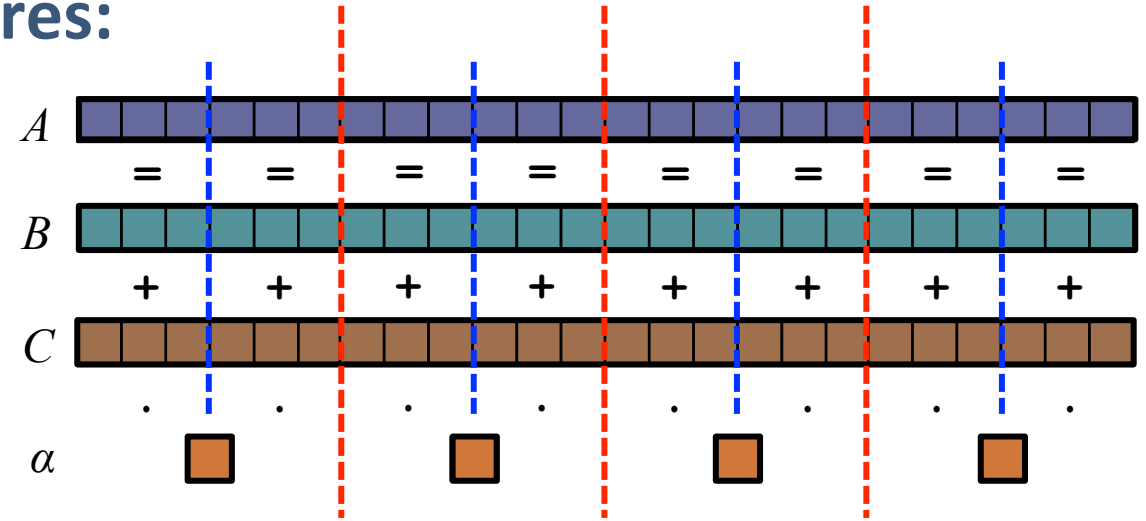
- Emerging parallel language
  - Open source initiative under development
  - Pioneered by Cray Inc.
    - In the context of DARPA's High Productivity Computing Systems (HPCS).
- Goals:
  - Productivity: performance and programmability
  - Portability
  - Robustness
  - Multiresolution philosophy

Chapel Motivation

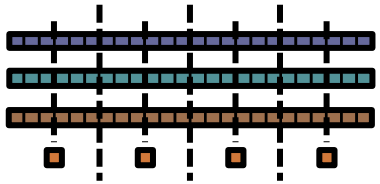
Given: m-element vectors A, B, C

Compute: $\forall i \in 1..m, \quad A_i = B_i + \alpha \cdot C_i$

In pictures:



MPI



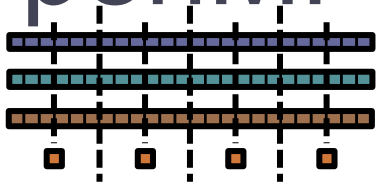
```
#include <hpcc.h>

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

int HPCC_StarStream(HPCC_Params *params) {
  int myRank, commSize;
  int rv, errCount;
  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;
}
```

```
for (j=0; j<VectorSize; j++) {
  b[j] = 2.0;
  c[j] = 3.0;
}

scalar = 3.0;
```

```
for (j=0; j<VectorSize; j++)
  a[j] = b[j]+scalar*c[j];

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

return 0;
}
```

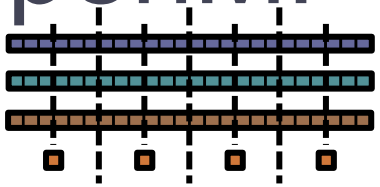
MPI + OpenMP



```
#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;
  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] = 3.0;
  }

  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);

  return 0;
}
```

MPI + OpenMP vs Cuda

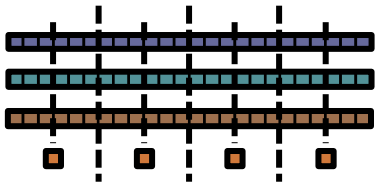
MPI + OpenMP



```
#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;
  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
  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] = 3.0;
  }
  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);

  return 0;
}
```

Where is programmer productivity ??

CUDA



```
#define N       2000000

int main() {
  float *d_a, *d_b, *d_c;
  float scalar;

  cudaMalloc((void**)&d_a, sizeof(float)*N);
  cudaMalloc((void**)&d_b, sizeof(float)*N);
  cudaMalloc((void**)&d_c, sizeof(float)*N);

  dim3 dimBlock(128);
  dim3 dimGrid(N/dimBlock.x );
  if( N % dimBlock.x != 0 ) dimGrid
                                    >(d_b, 2.0f, N);
                                    >(d_c, 3.0f, N);

                                  k>>>(d_b, d_c, d_a, scalar,  N);
  cudaThreadSynchronize();

  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);
}

__global__ void set_array(float *a,  float value, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) a[idx] = value;
}

__global__ void STREAM_Triad( float *a, float *b, float *c,
                              float scalar, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) c[idx] = a[idx]+scalar*b[idx];
}
```

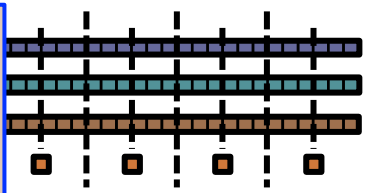
MPI + OpenMP vs Cuda

MPI + Ope

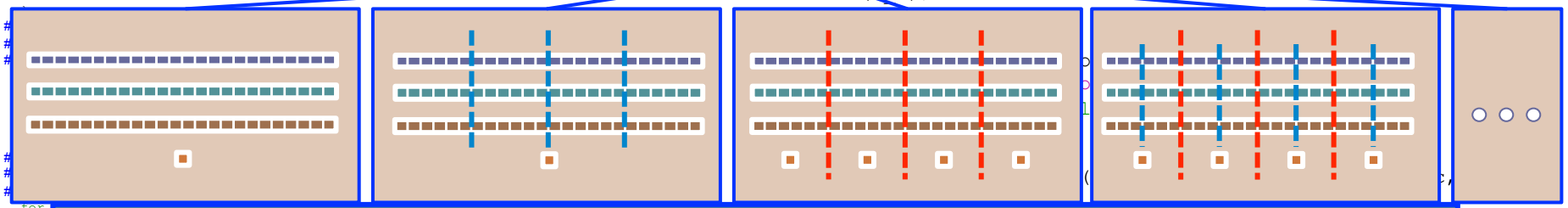
```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif
static int VectorSize;
static double *a, *b, *c;
int HPCC_StarStream(HPCC_Params *params)
  int myRank, commSize;
  int rv, errCount;
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;
  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );
  rv = HPCC_Stream( params, 0 == myRank);
  MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI
  return errCount;
}
int HPCC_Stream(HPCC_Params *params, int d
  register int j;
  double  scalar;
  VectorSize = HPCC_LocalVectorSize( params,
  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 a
      fclose( outFile );
    }
    return 1;
```

Chapel

```
config const m = 1000,
             alpha = 3.0;

const ProblemSpace = [1..m] dmapped …;

var A, B, C: [ProblemSpace] real;

B = 2.0;
C = 3.0;

A = B + alpha * C;
```



```
N);
N);
N);

0f, N);
0f, N);

d_c, d_a, scalar,  N);

cudaFree(d_c);
```

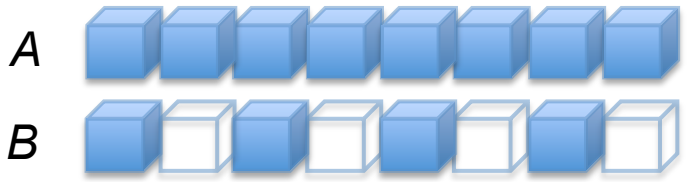


Philosophy: Good language design can tease details of locality and parallelism away from an algorithm, permitting the compiler, runtime, applied scientist, and HPC expert each to focus on their strengths.

Chapel Background

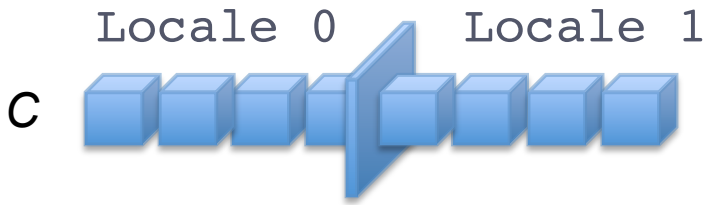
- Domains: index space

```
var DA = [1..8];
var DB = [1..8 by 2];
var A:[DA] real;
var B:[DB] real;
```



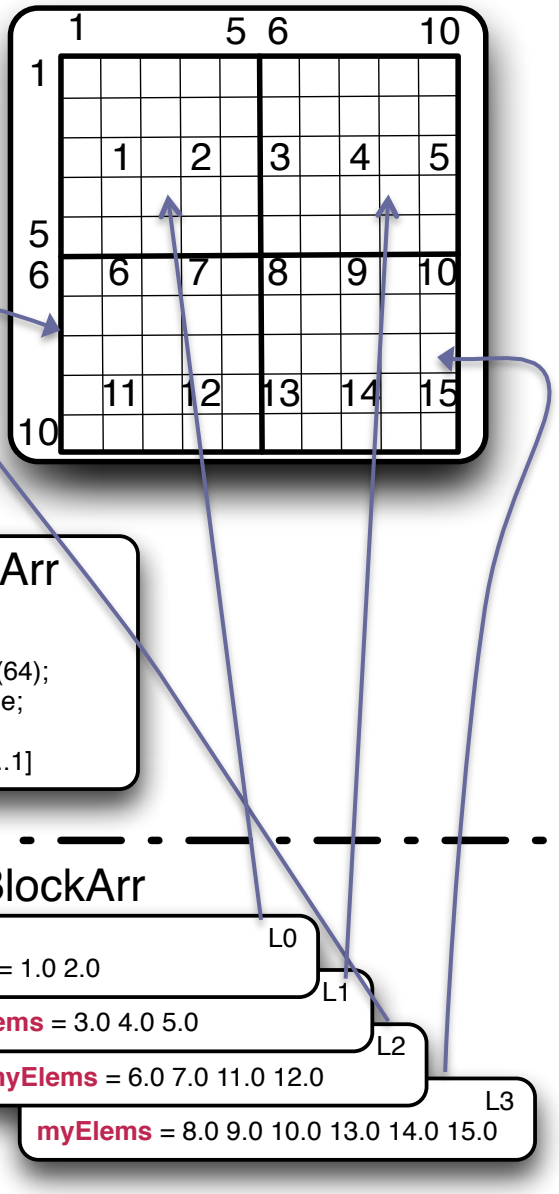
- Data distributions:
  - User defined distributions
  - Standard distributions: Block, Cyclic, Block-Cyclic, ...

```
var DC = DA dmapped Block(DA);
var C:[DC] real;
```



Chapel

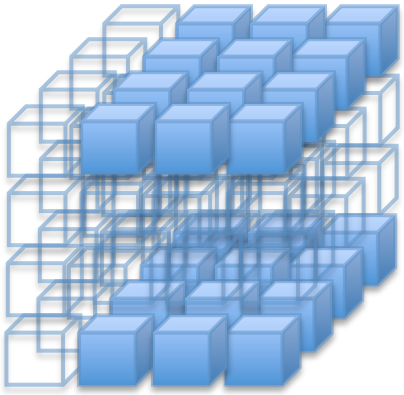
```
BBox = [1..10,1..10];
Dom = [3..10 by 3, 2..10 by 2];
const myDist = new dmap(new Block(BBox));
const myDom = Dom dmapped myDist;
var A:[myDom] real(64) = 1..;
```



GASNet

[1..4, 1..4 by 3, 2..4]

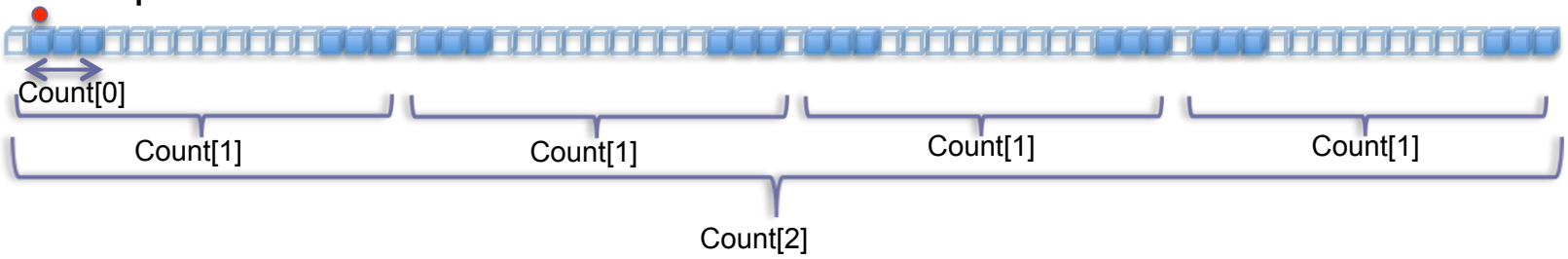
3D Representation



void **gasnet_gets_bulk** (DstAddr, DstStrides, SrcNode, SrcAddr, **SrcStrides**, **Count** , **StrideLevels**)

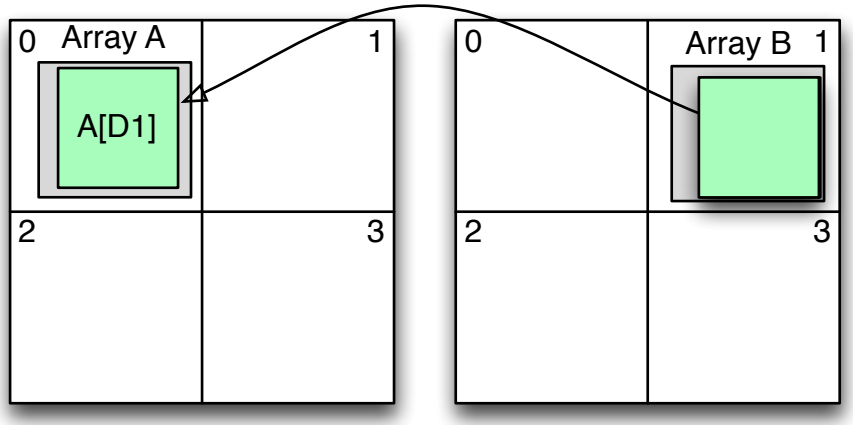
void **gasnet_puts_bulk**(DstNode, DstAddr, DstStrides, SrcAddr, **SrcStrides**, **Count**, **StrideLevels**)

1D Representation



StrideLevels = 2 Count=(3,2,4) SrcStrides= (12, 16)

Data aggregation implementation

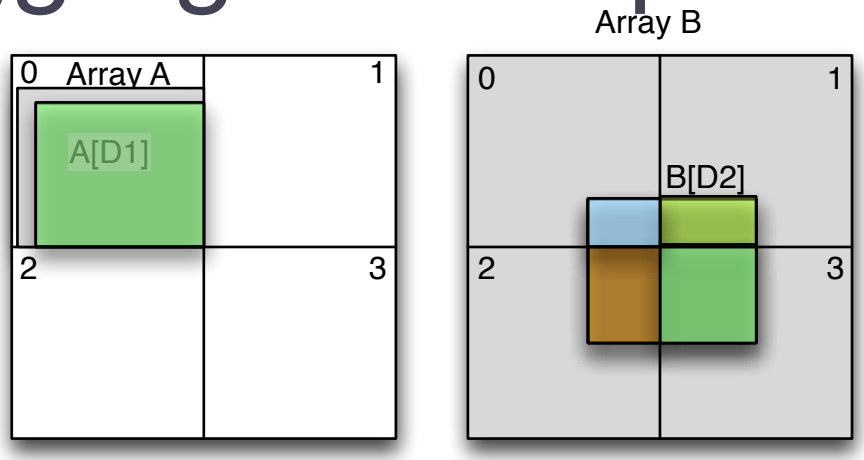


A[D1] = B[D2]

DR = Default Rectangular
(a C array with meta-info)

- DR = DR
  - A and B are DR arrays, A allocated on locale 0 and B on locale 1.
  - One call to `gasnet_gets_bulk` (if executed on locale 0) or `gasnet_puts_bulk` (if executed on locale 1)

Data aggregation implementation



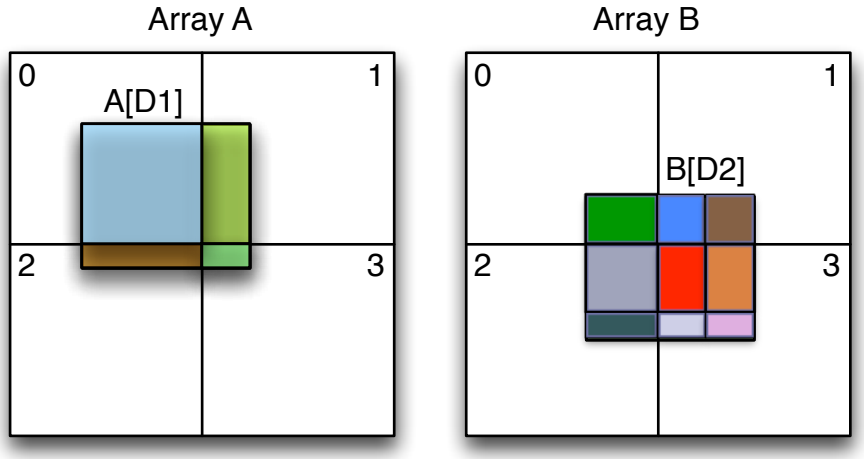
A[D1] = B[D2]

- DR = BD

BD = Block Distributed Array
(rely on several DR arrays)

```
forall i in 0..3 do
  on A do  { //co-locate with A
     // Run DR=DR assignments in parallel
     A[Slice_i] = B.locArr[i].myElems
  }
```

Data aggregation implementation



A[D1] = B[D2]

- BD = BD

```
forall i in 0..3 do
  on A.locArr[i] do { // co-locate with locArr[i]
     // Run DR=BD assignments in parallel
     A.locArr[i].myElems[dest] = B[Slice_i];
  }
```

Results

Cray HECToR supercomputer:

- Cray XE6, 90,112 cores - AMD Opteron 2.3 GHz.
- Cray Gemini interconnection network.
- Peak capacity: 800 Tflop/s.
- Top500.org: 19

Cray Jaguar supercomputer:

- Cray XK6, 298,592 cores - AMD Opteron 2.2 GHz.
- Gemini 3D torus interconnection network.
- Peak capacity: 2.6 Pflop/s.
- Top500.org: 6

Results
(BlockDist re-allocation)

```
config const n = 500;

var Dist1 =  new dmap(new Block(
var Dist2 =  new dmap(new Block(

var Dom1: domain(3, int) dmapped
var Dom2: domain(3, int) dmapped

var A:[Dom1] real(64);
var B:[Dom2] real(64);

var D=[1..n,1..n by 4,1..n];

A[D]=B[D]; // Assignment
```

Two Block distributions

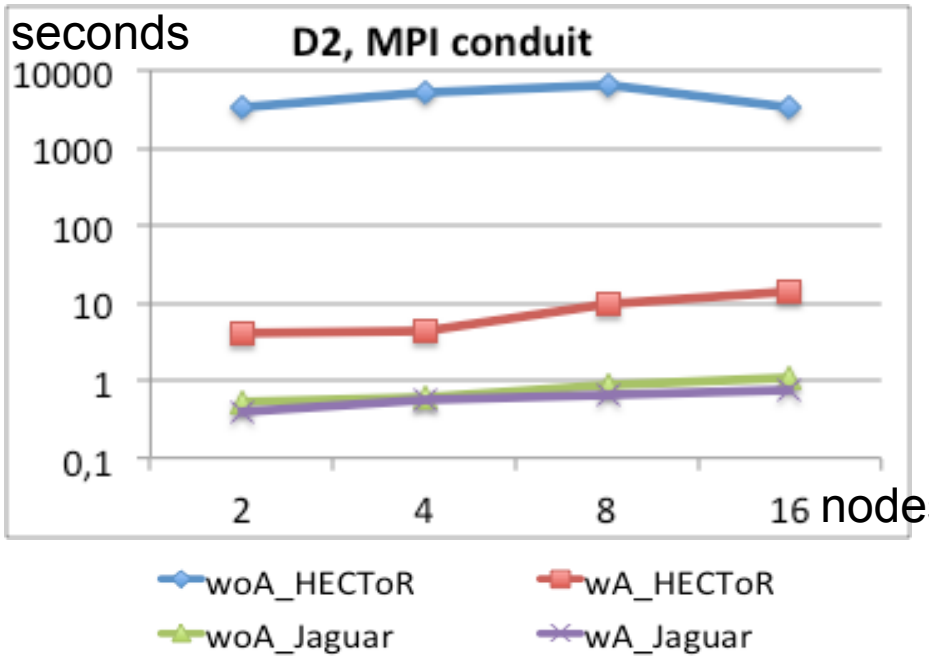
Two 3D distributed domains

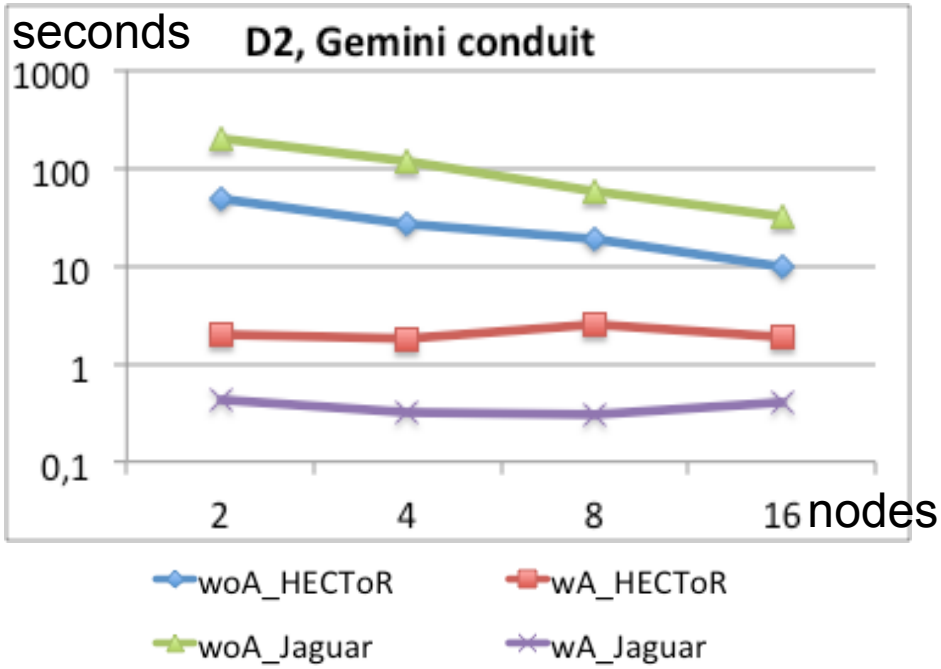
Two 3D distributed ARRAYS

Slice domain: 500x125x500 indices

Array assignment: 250 MBytes

Results
(BlockDist re-allocation)





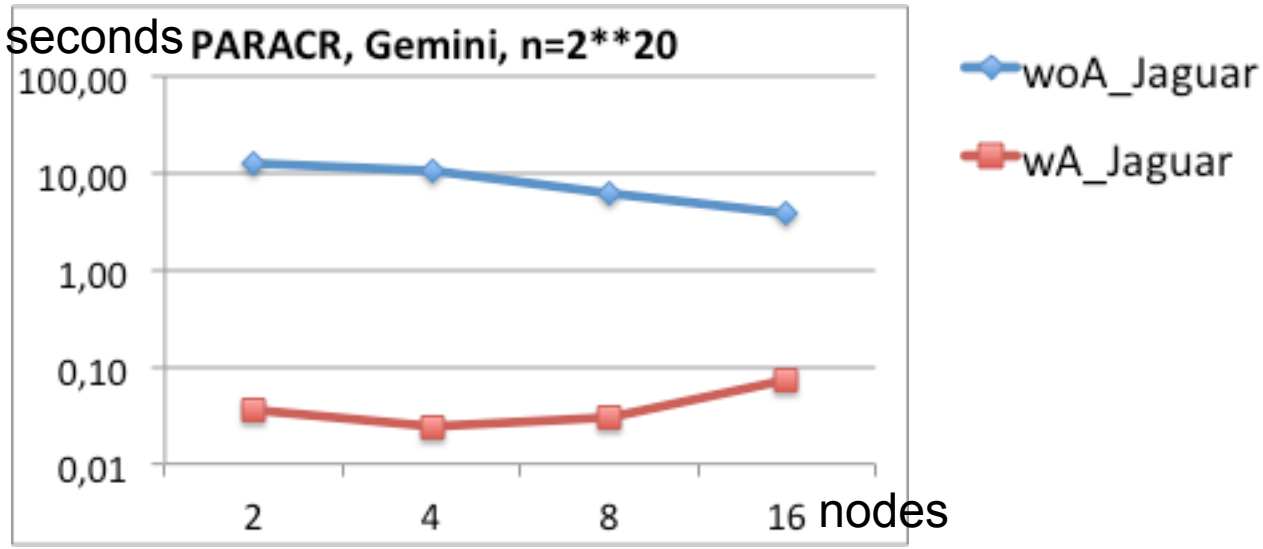
- wA - with Aggregation optimization, speedup vs. woA up to 1,000x
- Gemini conduit is faster than MPI conduit on each machine

Results
(FFT, Block to Cyclic redistribution)

FFT on **HECToR**, Gemini conduit, $n=2^{20}$					
	Without aggregation		With aggregation		
Loc	T. BtoC	T. CtoB	T. BtoC	T. CtoB	Comm. Imp
1	0,850	6,6653	0,0109	0,0099	**361,31**
4	0,944	5,4038	0,3032	1,3381	**4,63**
16	2,143	5,1227	0,2454	0,4146	**11,00**
FFT on **Jaguar**, Gemini conduit, $n=2^{20}$					
	Without aggregation		With aggregation		
Loc	T. BtoC	T. CtoB	T. BtoC	T. CtoB	Comm. Imp
1	0,0175	0,0761	0,00495	0,03987	**2,09**
4	3,1547	4,6204	0,01095	0,08519	**80,88**
16	1,3238	1,5435	0,03310	0,05919	**31,07**

- Part of HPC Challenge (HPCC) suite
- Aggregation speedup up to 80x

Results
(PARACR, Block to Cyclic redistribution)



- PARACR is an algorithm for solving tridiagonal systems of equations
- Block distribution better for the first steps, Cyclic for the last steps
- Aggregation speedup for Block-to-Cyclic redistribution up to 1,000x

Conclusions

- Chapel is an emerging parallel programming language
- This work explores the aggregation of communications in Block and Cyclic Distribution
- We take advantage of GASNet one-sided bulk communication routines
- The results show significant speedups for communications times
- Included in the new release: Chapel 1.6

- Future Work:
  - Generalize this optimization to other distributions
  - Improve scheduling of communications

For More Information

Chapel project page:

http://chapel.cray.com

- Overview, papers, presentations, spec, …

Chapel SourceForge page:

https://sourceforge.net/projects/chapel/

- Release downloads, public mailing lists, code repository, …

Research group page:

http://www.ac.uma.es/~asenjo/research/

Parallel programming models, Parallel languages (Chapel, UPC, X10), Parallel libraries (TBB), …