Multiresolution Parallel Programming with Chapel

Vassily Litvinov and the Chapel team, Cray Inc.

HPC Advisory Council Conference

13 September 2012

Outline

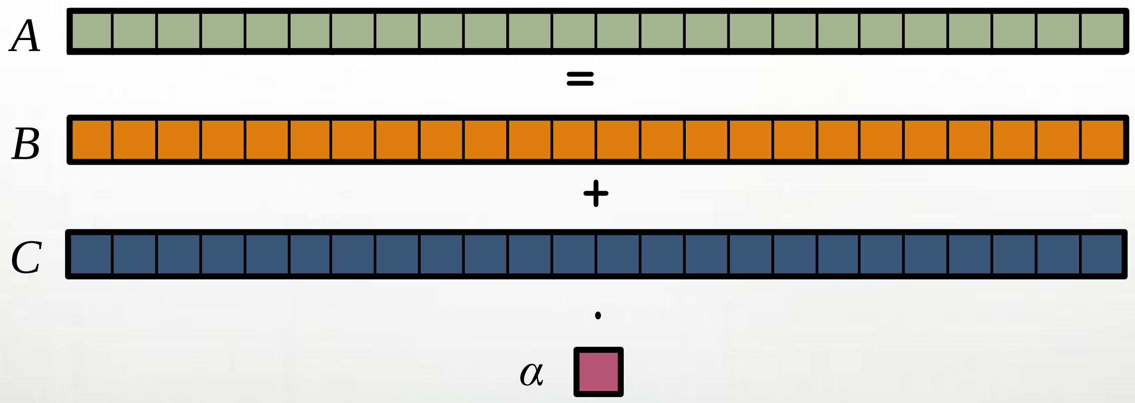
- ➢ Motivation: Programming Models
- Multiresolution Programming
- Empirical Evaluation
- About the Project

STREAM Triad: a trivial parallel computation

Given: m-element vectors A, B, C

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

In pictures:

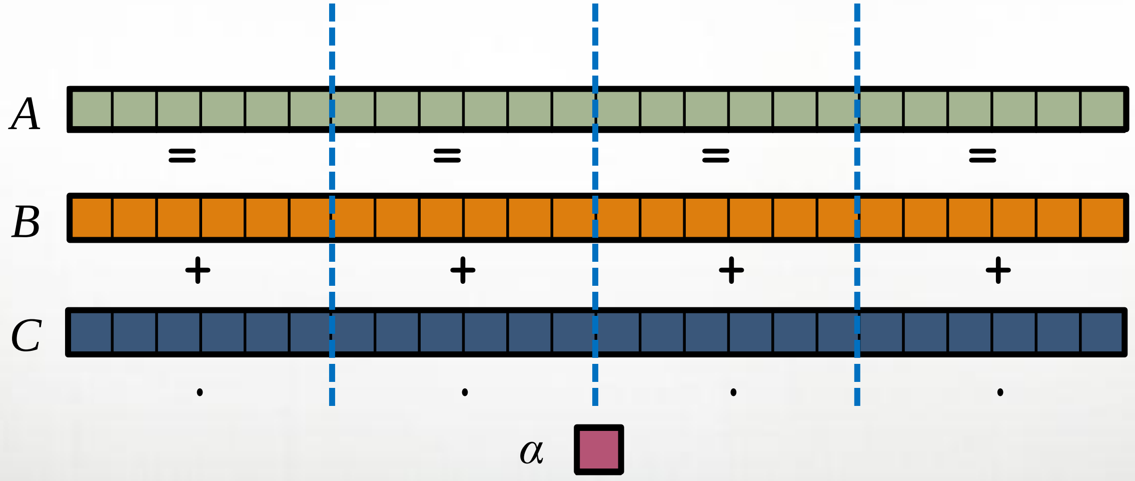


STREAM Triad: a trivial parallel computation

Given: m-element vectors A, B, C

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

In pictures, in parallel:

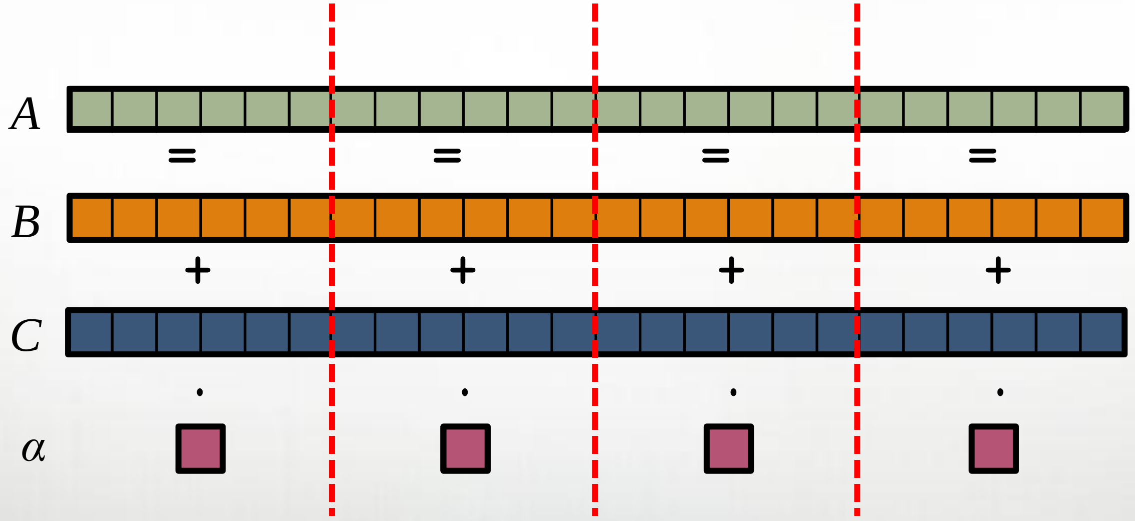


STREAM Triad: a trivial parallel computation

Given: m-element vectors A, B, C

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

In pictures, in parallel, distributed memory:

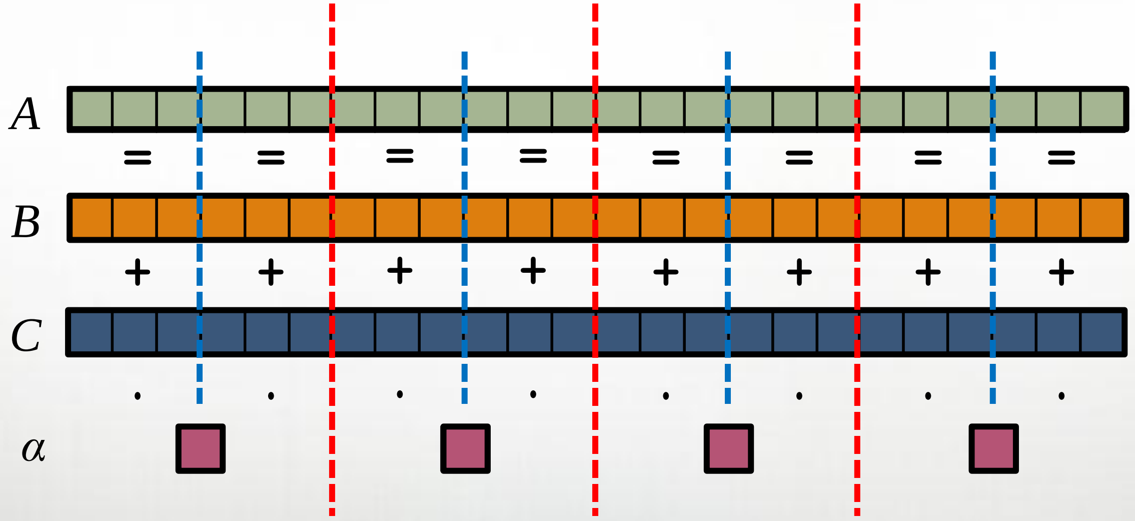


STREAM Triad: a trivial parallel computation

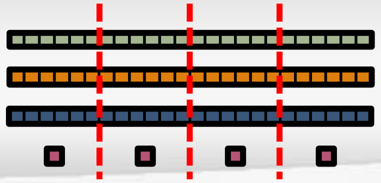
Given: m-element vectors A, B, C

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

In pictures, in parallel, distributed memory, multicore:



STREAM Triad: MPI

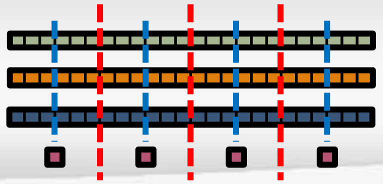


MPI

```
#include <hpcc.h>




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;
  }




  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;
}
```

STREAM Triad: MPI+OpenMP



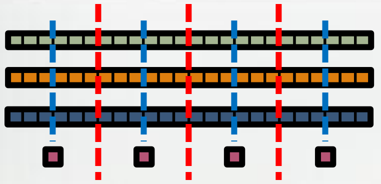
MPI + OpenMP

```
#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;
  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;
}
```

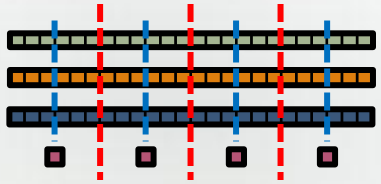
STREAM Triad: MPI+OpenMP vs. CUDA

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;
  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;
}
```

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.x+=1;

  set_array<<<dimGrid,dimBlock>>>(d_b, 2.0f, N);
  set_array<<<dimGrid,dimBlock>>>(d_c, 3.0f, N);

  scalar=3.0f;
  STREAM_Triad<<<dimGrid,dimBlock>>>(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];
}
```

Why so many programming models?

Examples:

Type of HW Parallelism	Programming Model	Unit of Parallelism
Inter-node	MPI	process
Intra-node/multicore	OpenMP/pthreads	iteration/task
Instruction-level vectors/threads	pragmas	iteration
GPU/accelerator	CUDA/OpenCL/OpenAcc	SIMD function/task

HPC has traditionally given users…

…low-level, *control-centric* programming models

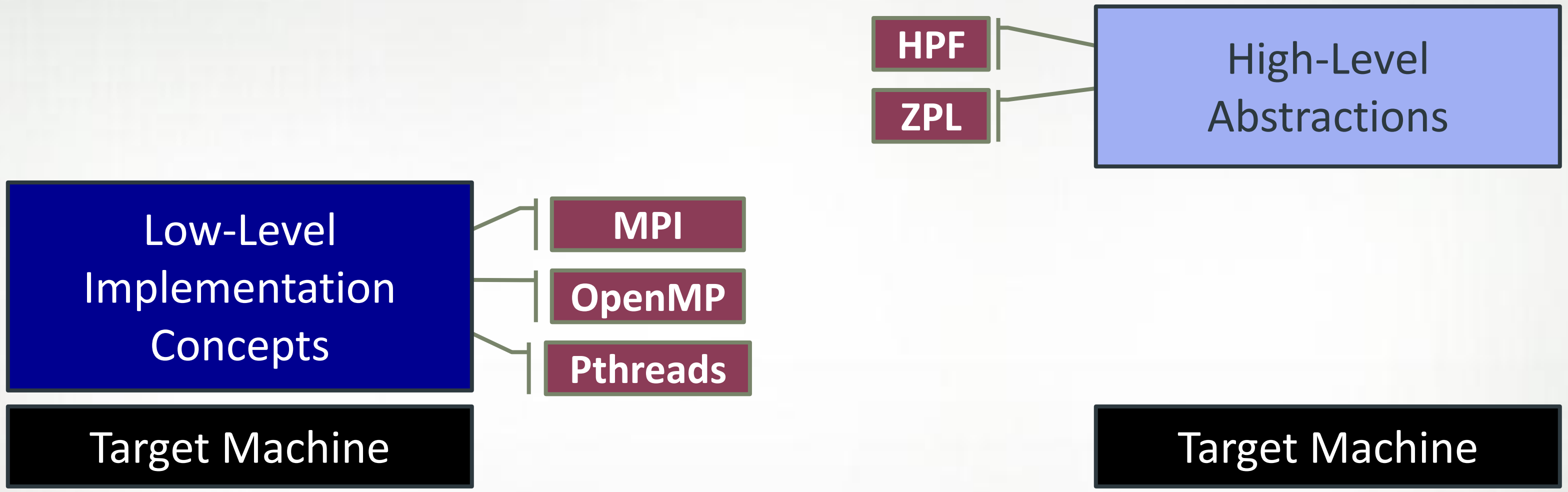
…ones that are closely tied to the underlying hardware

…ones that support only a single type of parallelism

benefits: lots of control; decent generality; easy to implement

downsides: lots of user-managed detail; brittle to changes

Multiresolution Design: Motivation



"Why is everything so tedious/difficult?"
"Why don't my programs port trivially?"

"Why don't I have more control?"

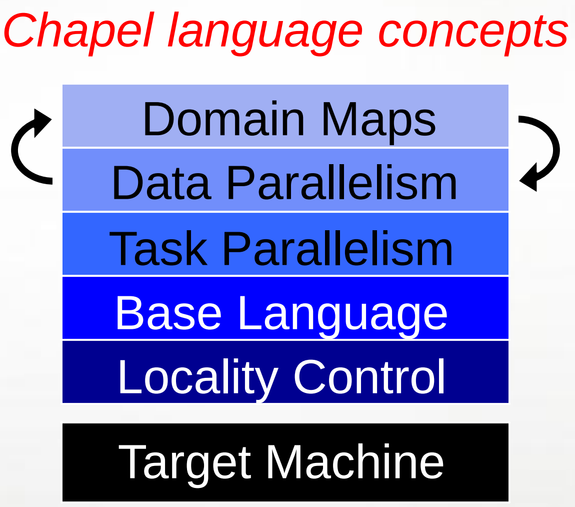
Outline

- ✓ Motivation
- ➢ Multiresolution programming
- Empirical Evaluation
- About the Project

Multiresolution Design Philosophy

Multiresolution Design: Support multiple tiers of features

- higher levels for programmability, productivity
- lower levels for greater degrees of control



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.

STREAM Triad: MPI+OpenMP vs. CUDA

MPI + OpenMP

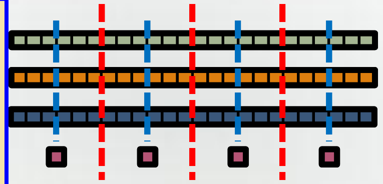
```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

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

int HPCC_StarStream(HPCC_Params *par
  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 == myR
  MPI_Reduce( &rv, &errCount, 1, MPI

  return errCount;
}

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

  VectorSize = HPCC_LocalVectorSize(

  a = HPCC_XMALLOC( double, VectorSi
  b = HPCC_XMALLOC( double, VectorSi
  c = HPCC_XMALLOC( double, VectorSi

  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
```

```
  }

  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;
}
```

Chapel

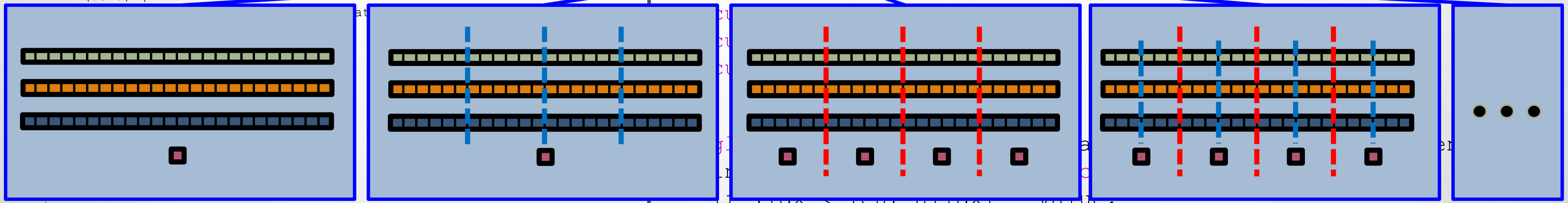
```
config const m = 1000,
             alpha = 3.0;

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

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

B = 2.0;
C = 3.0;

A = B + alpha * C;
```

the special sauce



```
;
;
;
N);
N);
d_c, d_a, scalar,  N);
```

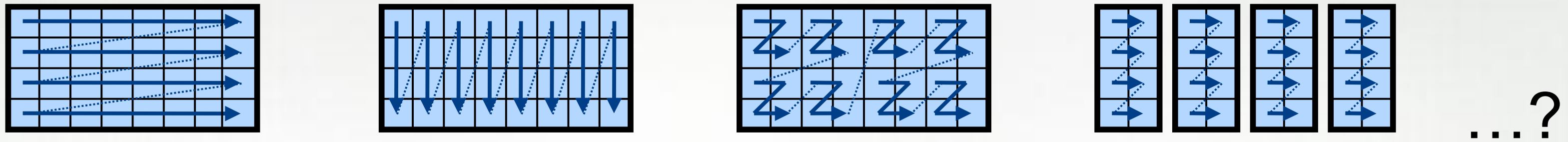
```
}

__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];
}
```



Data Parallelism Implementation Qs

Q1: How are arrays laid out in memory?

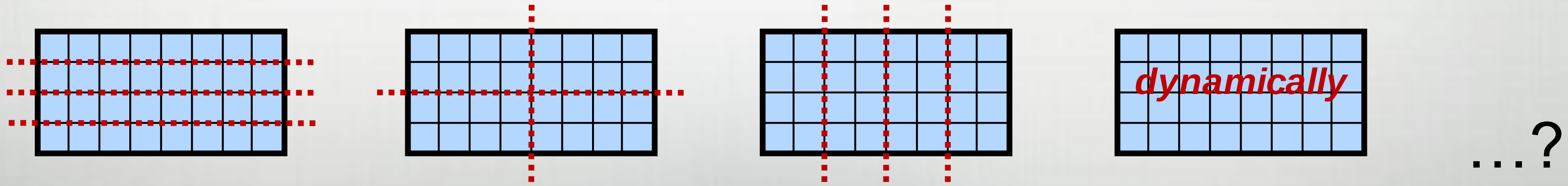
- Are regular arrays laid out in row- or column-major order? Or…?



- How are sparse arrays stored? (COO, CSR, CSC, block-structured, …?)

Q2: How are arrays stored by the locales?

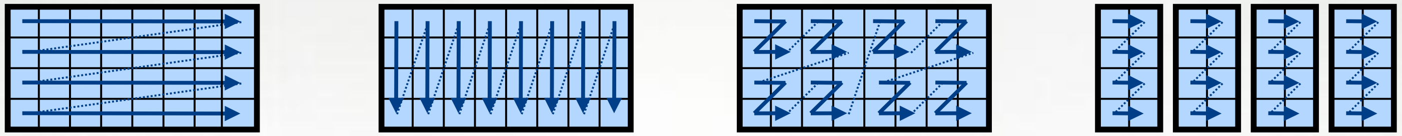
- Completely local to one locale? Or distributed?
- If distributed… In a blocked manner? cyclically? block-cyclically? recursively bisected? dynamically rebalanced? …?



Data Parallelism Implementation Qs

Q1: How are arrays laid out in memory?

- Are regular arrays laid out in row- or column-major order? Or…?



…?

- How are sparse arrays stored? (COO, CSR, CSC, block-structured, …?)

Q2: How are arrays stored by the locales?

- Completely local to one locale? Or distributed?
- If distributed… In a blocked manner? cyclically? block-cyclically? recursively bisected? dynamically rebalanced? …?

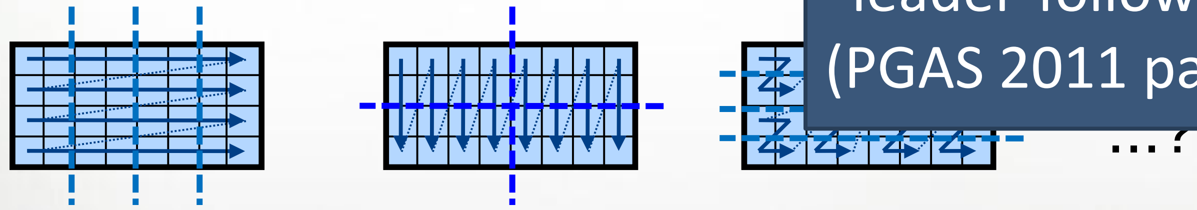
A: Chapel's *domain maps* are designed to give the user full control over such decisions

More Data Parallelism Implementation Qs

Q: How are loops implemented?

```
A = B + alpha * C;    // an implicit loop
```

- How many tasks? Where do they execute?
- How is the iteration space divided between the tasks?
  - statically? dynamically? what algo

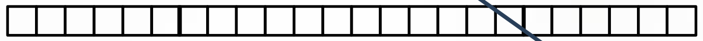


"leader-follower" iterators (PGAS 2011 paper)

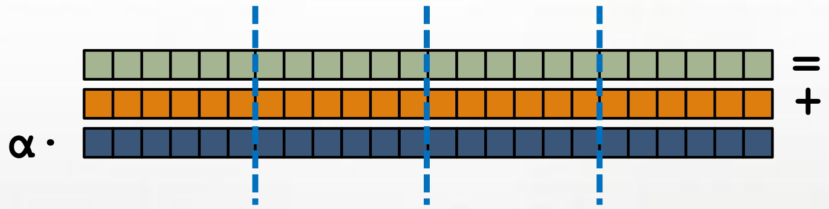
A: Chapel's domain maps are designed to give the user full control here, too

STREAM Triad: Chapel (multicore)

```
const ProblemSpace = [1..m];
```



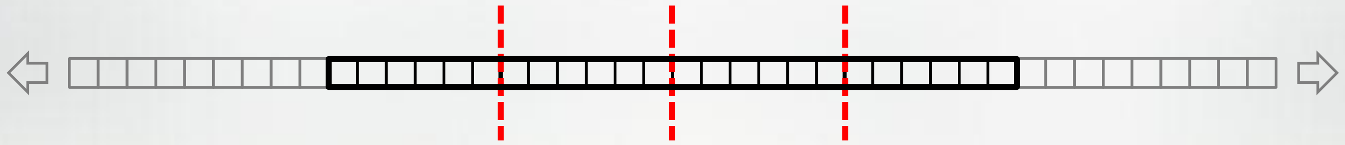
```
var A, B, C: [ProblemSpace] real;
```



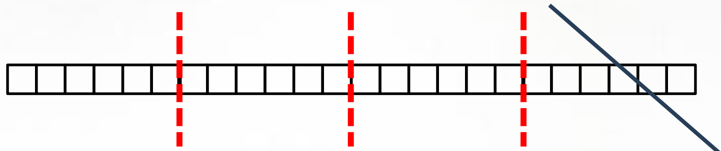
```
A = B + alpha * C;
```

No domain map is specified => use the default one
- current locale owns all indices and values
- computation will execute using local processors only, in parallel

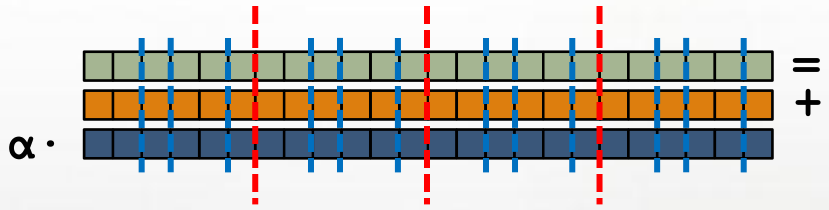
STREAM Triad: Chapel (multilocale, blocked)



```
const ProblemSpace = [1..m];
                  dmapped Block(boundingBox=[1..m]);
```



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

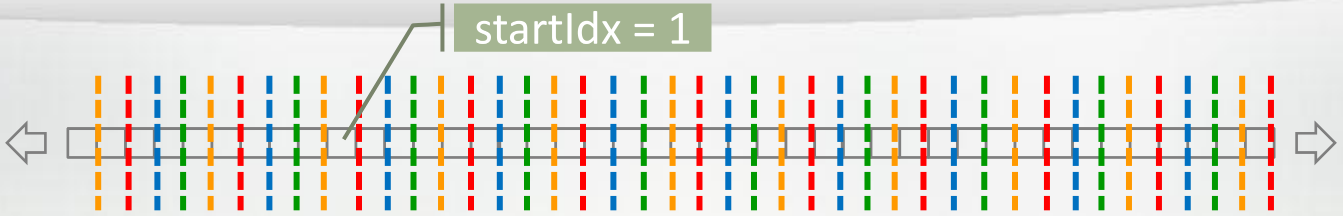


```
A = B + alpha * C;
```

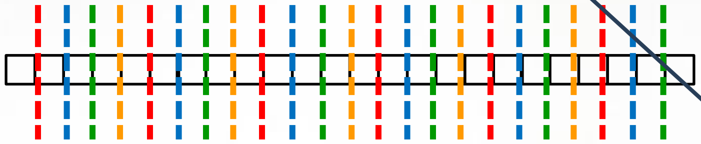
Block domain map is chosen explicitly

- indices and values are distributed over all locales
- computation will execute on all locales and processors, in parallel

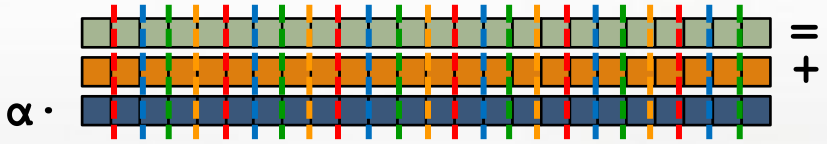
STREAM Triad: Chapel (multilocale, cyclic)



```
const ProblemSpace = [1..m];
                               dmapped Cyclic(startIdx=1);
```



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



```
A = B + alpha * C;
```

Cyclic domain map is chosen explicitly

- similarly, distributed values, distributed+parallel computation

Domain Maps: Some Details

- Given an implicit loop…

```
A = B + alpha * C;
```

- or an equivalent explicit loop
  - **`forall`** indicates it is parallel

```
forall (a,b,c) in (A,B,C) {
  a = b + alpha * c;        }
```

Chapel's iterator – here enables user to introduce distribution and parallelism

- the compiler converts it to

```
for followThis in A.domain_map.these(...) {
  for (a,b,c) in (A.domain_map.these(followThis,...),
                  B.domain_map.these(followThis,...),
                  C.domain_map.these(followThis,...)) {
    a = b + alpha * c;                              } }
```

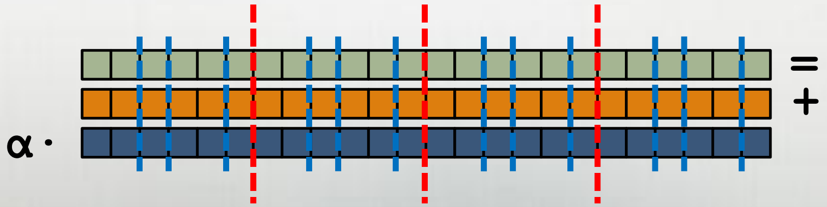
"leader/follower" scheme (not in this talk)

pseudocode

Domain Maps: The User Can

- … and the domain map author implements `these` iterators, for example:

```
iter MyDomainMap.these(...) {
  coforall loc in Locales {
    on loc {
      coforall task in 1..here.numCores {
        yield computeMyChunk(loc.id, task);
      }
  } }
}
```



Chapel's Domain Map Philosophy

Domain maps are "recipes" that instruct the compiler how to implement global-view computations

- Unless requested explicitly, a reasonable default domain map/ implementation is used
- Chapel provides a library of standard domain maps
  - to support common array implementations effortlessly
- Advanced users can write their own domain maps in Chapel
  - to cope with shortcomings in the standard library
  - using Chapel – all of the language is fully available

switching to other resolution levels for more control

• not required, but available when desired

Multiresolution Design: Summary

- Chapel avoids locking crucial implementation decisions into the language specification
  - local and distributed array implementations
  - parallel loop implementations

- Instead, these can be…

  …specified in the language by an advanced user

  …swapped in and out with minimal code changes

- The result cleanly separates the roles of domain scientist, parallel programmer, and implementation

Outline

- ✓ Motivation
- ✓ Multiresolution programming
- ➢ Empirical Evaluation
- About the Project

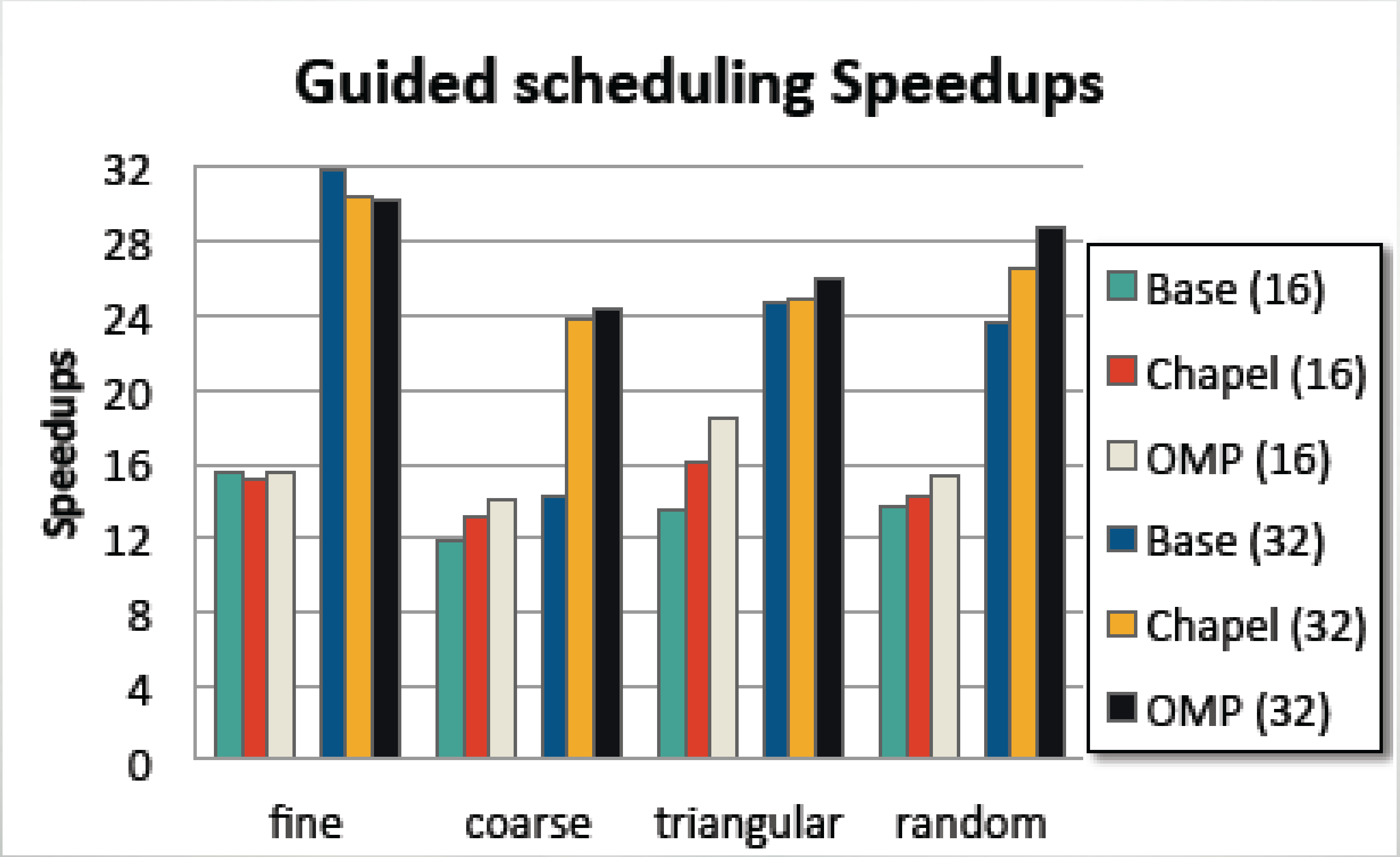
User-Defined Parallel Iterators

PGAS 2011: *User-Defined Parallel Zippered Iterators in Chapel,* Chamberlain, Choi, Deitz, Navarro; October 2011

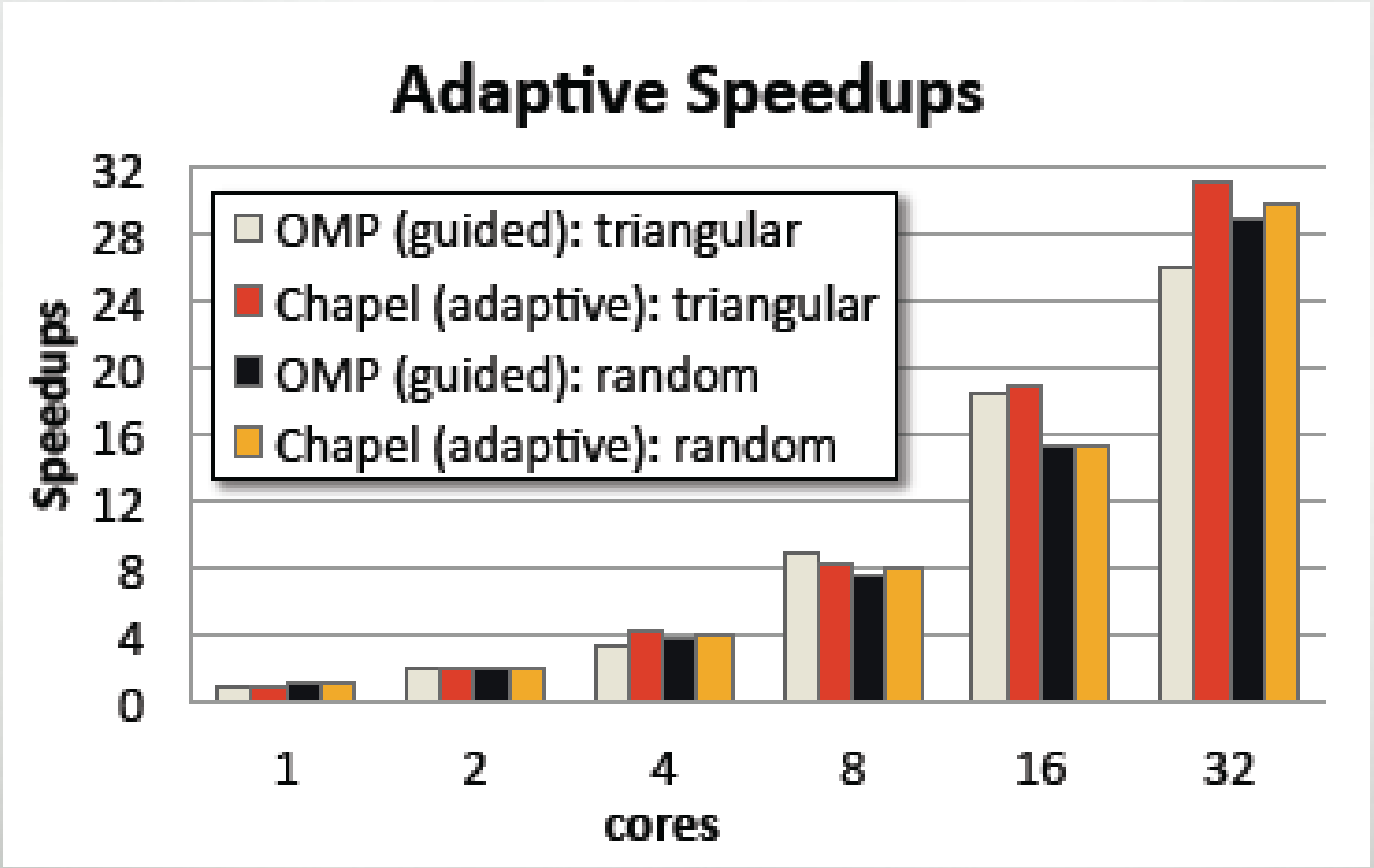
- Implemented various scheduling policies
  - OpenMP-style dynamic and guided
  - adaptative, with work stealing
  - available as iterators
- Compared performance against OpenMP
  - Chapel is competitive

Chapel's multi-resolution design allows HPC experts to implement desired policies and scientists to incorporate them with minimal code changes

Chapel vs. OpenMP Guided



Chapel Adaptive vs. OpenMP Guided



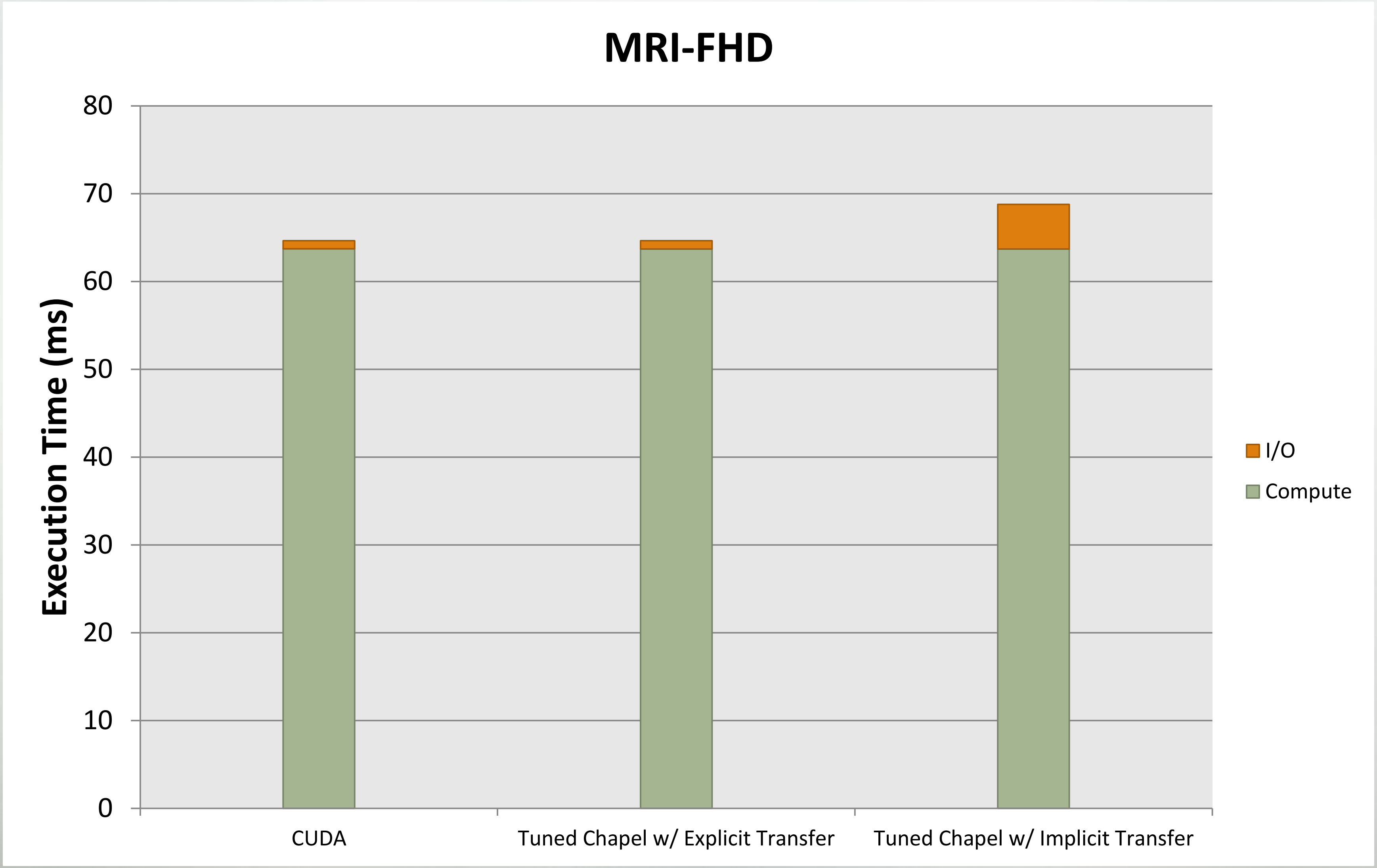
Targetting GPUs

IPDPS 2012: *Performance Portability with the Chapel Language,*
Sidelnik, Maleki, Chamberlain, Garzarán, Padua; May 2012

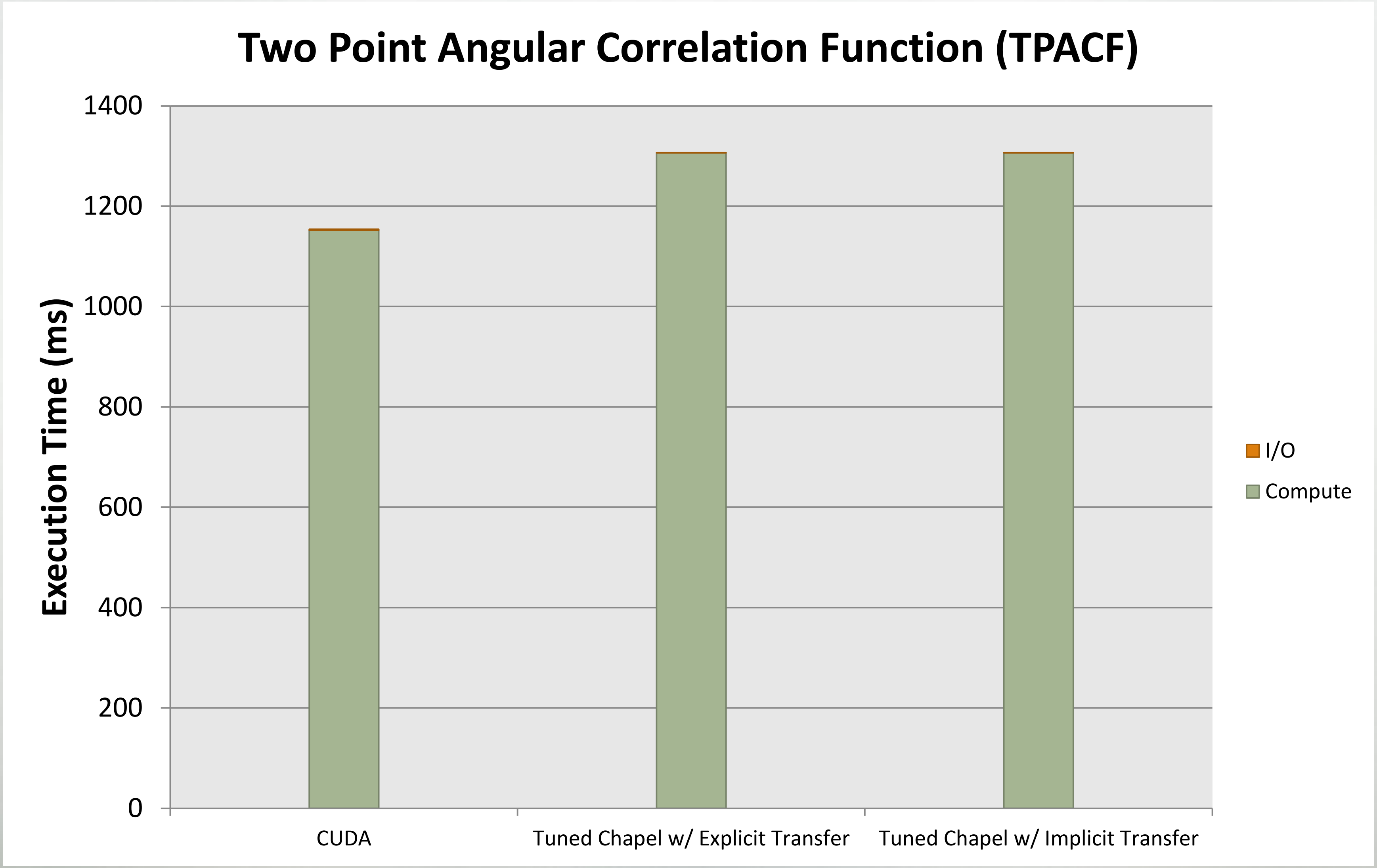
- Technology for running Chapel code on GPUs
  - implemented a domain map to place data and execute code on GPUs
  - added compiler support to emit CUDA code; additional optimizations
- Compared performance against hand-coded CUDA
  - competitive performance, less code

The domain map allows the user to target GPUs with minimal code changes

Parboil Benchmark Suite



Parboil Benchmark Suite



Code Size Comparison

Benchmark	# Lines (CUDA)	# Lines (Chapel)	% difference	# of Kernels
CP	186	154	17	1
MRI-FHD	285	145	49	2
MRI-Q	250	125	50	2
RPES	633	504	16	2
TPACF	329	209	36	1

Outline

- ✓ Motivation
- ✓ Multiresolution programming
- ✓ Empirical Evaluation
- ➢ About the Project

Chapel's Implementation

- Being developed as open source at SourceForge
  - BSD license
- **Target Architectures:**
  - Cray architectures
  - multicore desktops and laptops
  - commodity clusters
  - systems from other vendors
  - *in-progress:* CPU+accelerator hybrids, manycore, …
- Try it out and give us feedback!

Some Next Steps

- Hierarchical Locales
- Resilience Features
- Performance Optimizations
- Evolve from Prototype- to Production-grade
- Evolve from Cray- to community-language
- and much more…

For More Information

Chapel project page: http://chapel.cray.com

- overview, papers, presentations, language spec, …

Chapel SourceForge page: https://sourceforge.net/projects/chapel/

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

Mailing Lists:

- chapel_info@cray.com: contact the team
- chapel-users@lists.sourceforge.net: user-oriented discussion list
- chapel-developers@lists.sourceforge.net: dev.-oriented discussion
- chapel-education@lists.sourceforge.net: educator-oriented discussion
- chapel-bugs@lists.sourceforge.net: public bug forum
- chapel_bugs@cray.com: private bug mailing list

http://chapel.cray.com   chapel_info@cray.com   http://sourceforge.net/projects/chapel/