Multiresolution Parallel Programming with Chapel

Vassily Litvinov and the Chapel team, Cray Inc.
University of Malaga – Department of Computer Architecture

11 September 2012

Outline

- Motivation: Programming Models
- Introducing Chapel
- Multiresolution Programming
- Empirical Evaluation
- About the Project

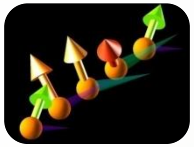
Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors

- Static finite element analysis
- Fortran77 + Cray autotasking + vectorization

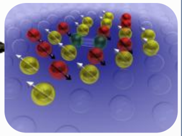

1 TF – 1998: Cray T3E; 1,024 Processors

- Modeling of metallic magnet atoms
- Fortran + MPI (Message Passing Interface)

1 PF – 2008: Cray XT5; 150,000 Processors

- Superconductive materials
- C++/Fortran + MPI + vectorization



1 EF – ~2018: Cray ____; ~10,000,000 Processors

- TBD
- TBD: C/C++/Fortran + MPI + CUDA/OpenCL/OpenMP/OpenACC

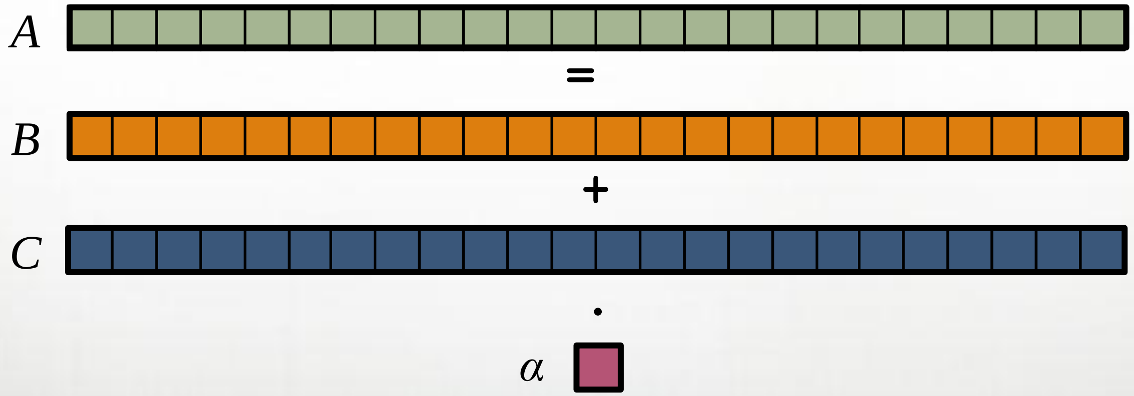
Or Perhaps Something Completely Different?

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:

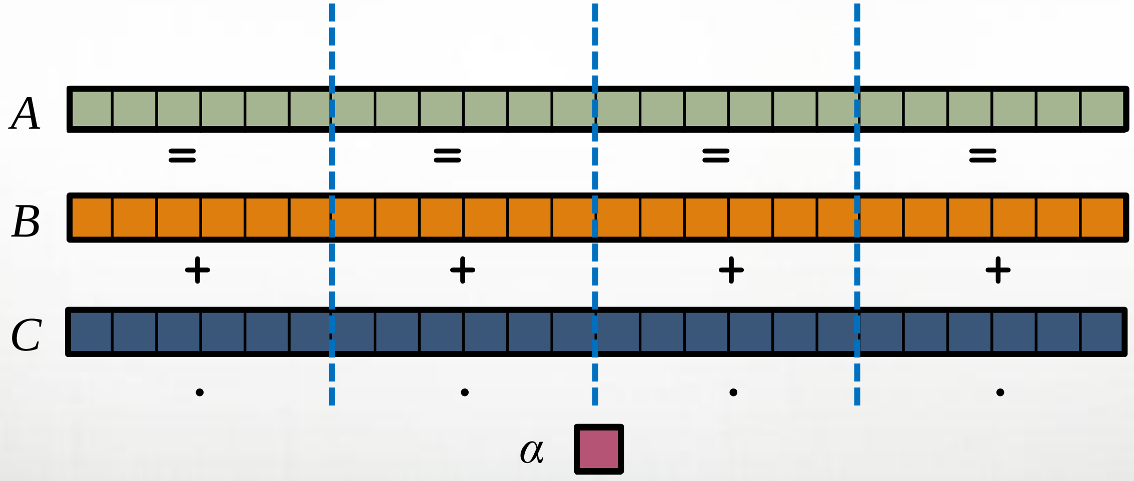


STREAM Triad: a trivial parallel computation

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:

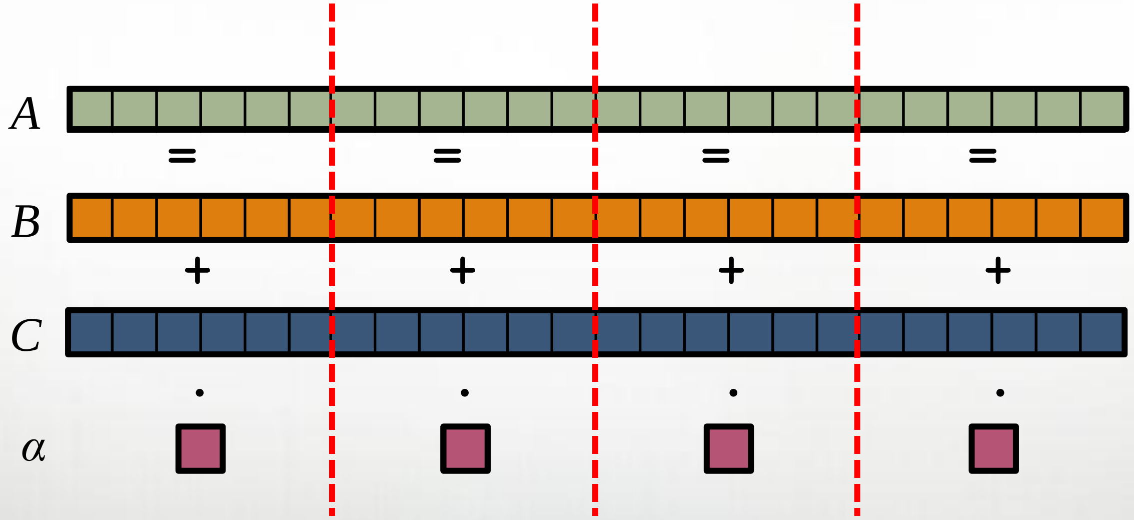


STREAM Triad: a trivial parallel computation

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:

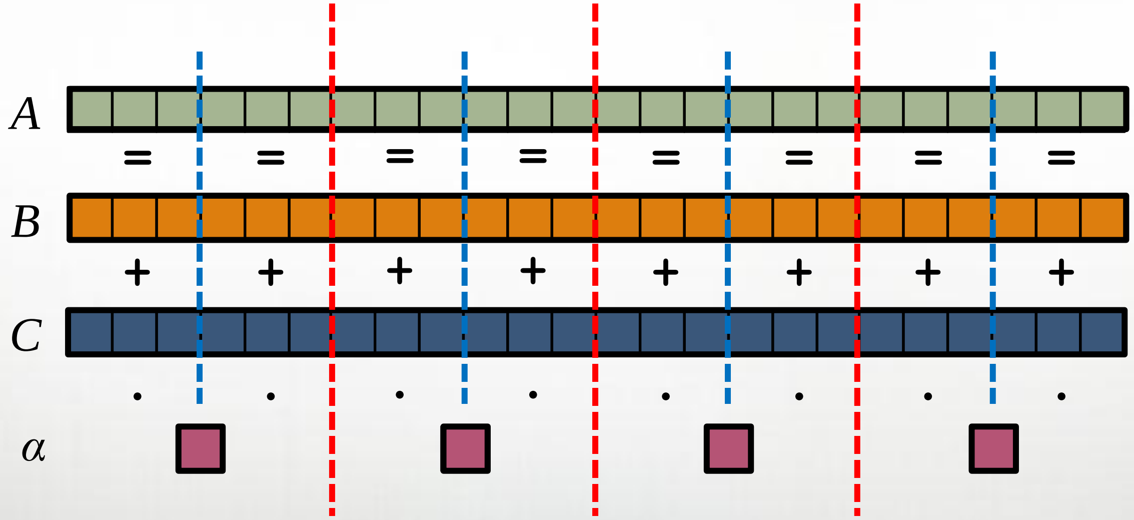


STREAM Triad: a trivial parallel computation

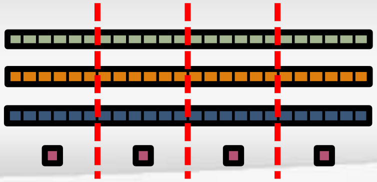
Given: m-element vectors A, B, C

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

In pictures, in parallel, distributed memory, multicore:



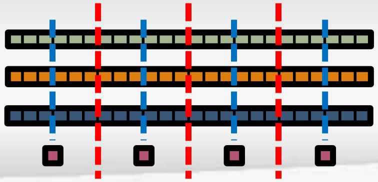
STREAM Triad: MPI



MPI

```c
#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



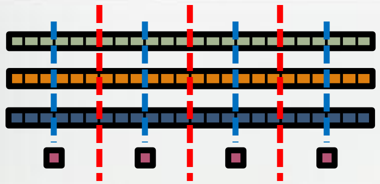
MPI + OpenMP

```
#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;
}
```

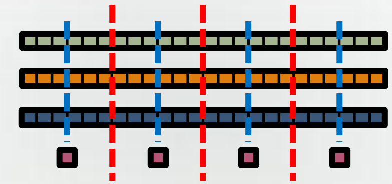
STREAM Triad: MPI+OpenMP vs. CUDA

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(

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

  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



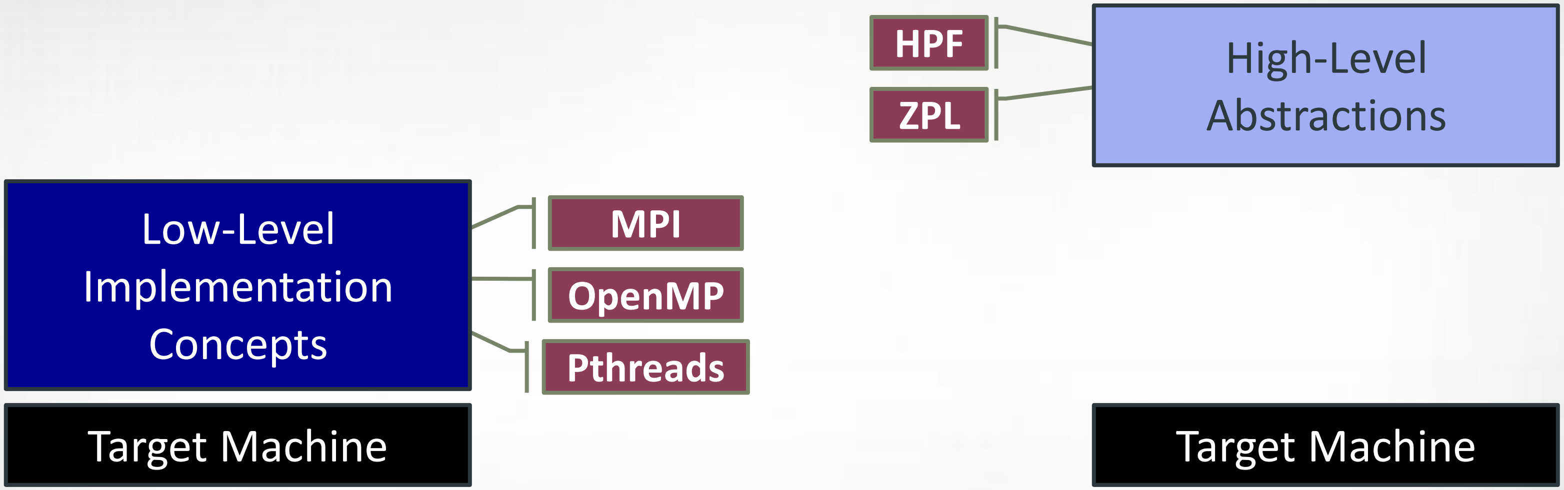
```
#define N        2000000

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;

ck>>>(d_b, 2.0f, N);
ck>>>(d_c, 3.0f, N);

  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];
}
```

Where is programmer productivity ??

By Analogy: Let's Cross the United States!

By Analogy: Let’s Cross the United States!

Multiresolution Design: Motivation



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

"Why don't I have more control?"

STREAM Triad: MPI+OpenMP vs. CUDA

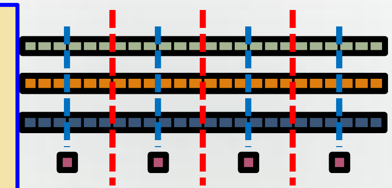
MPI + OpenMP

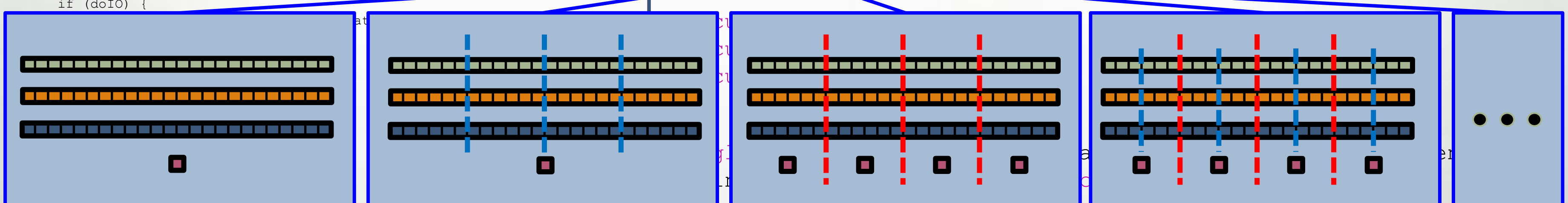
```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

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) {
```

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





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.

Outline

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

Chapel in a Nutshell

Chapel: a parallel language that has emerged from DARPA HPCS

- **general parallelism:**
  - data-, task-, and nested parallelism
  - highly dynamic multithreading or static SPMD-style
- **locality control:**
  - explicit or data-driven placement of data and tasks
  - locality expressed distinctly from parallelism
- **multiresolution philosophy:** high-level features built on low-level
  - to provide “manual overrides”
  - to support a separation of concerns (application vs. parallel experts)
- **features for productivity:** type inference, iterators, rich array types
- **portable:** designed and implemented to support diverse systems
- **open source:** developed and distributed under the BSD license
- **plausibly adoptable:** forward-thinking HPC users want a mature version

Static Type Inference

```
const pi = 3.14,              // pi is a real
      name = "vass";          // name is a string

var sum   = add(1, pi),                // sum is a real
    email = add(name, "@cray");    // email is a string

for i in 1..5 {
  if i == 2 { writeln(sum, " ", email); }
}

proc add(x, y) {              // add() is generic
  return x + y;               // its return type is inferred
}
```

```
4.14 vass@cray
```

Static Type Inference

```
const pi = 3.14,              // pi is a real
      name = "vass";          // name is a string

var sum   = add(1, pi),               // sum is a real
    email = add(name, "@cray");       // email is a string

for i in 1..5 {
  if i == 2 { writeln(sum, " ", email); }
}

proc add(x:int, y:real):real {        // or, explicit typing
  return x + y;                       // return expr is checked
}
proc add(x:string, y:string)  {       // need another overload
  return x + y;                // return type is inferred here
}
```

```
4.14 vass@cray
```

Iterators

```
iter fibonacci(n) {
  var current = 0,
      next = 1;
  for 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}
```

```
for f in fibonacci(7) do
  writeln(f);
```

```
0
1
1
2
3
5
8
```

Coforall Loops

```
coforall t in 0..#numTasks {
  writeln("Hello from task ", t, " of ", numTasks);
}
writeln("All tasks done");
```

```
Hello from task 2 of 4
Hello from task 0 of 4
Hello from task 3 of 4
Hello from task 1 of 4
All tasks done
```

Locality Control

```
writeln("on locale 0");

on Locales[1] do
  writeln("now on locale ", here.id);

writeln("on locale 0 again");
```

```
on locale 0
now on locale 1
on locale 0 again
```

Outline

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

Multiresolution Design: an Example

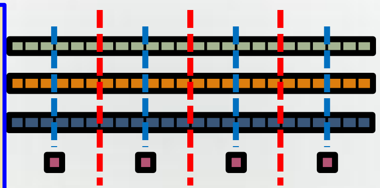
MPI + OpenMP

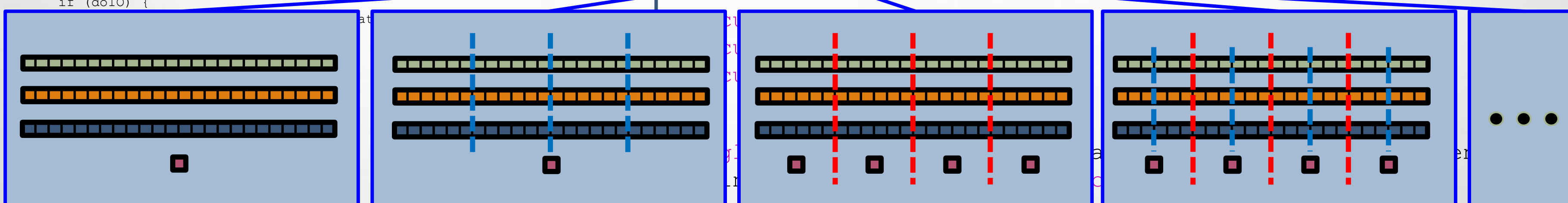
```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

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) {
```

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





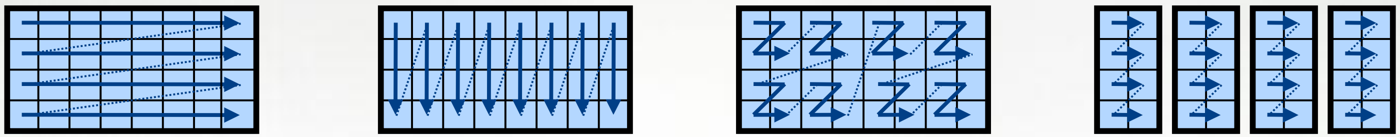
```
  scalar = 3.0;

#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;
```

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

  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];
}
```

Data Parallelism Implementation Qs

Q1: How are arrays laid out in memory?

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

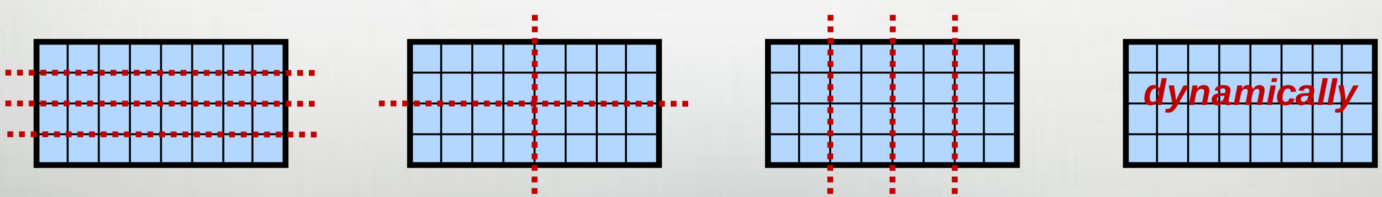


…?

- 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? …?

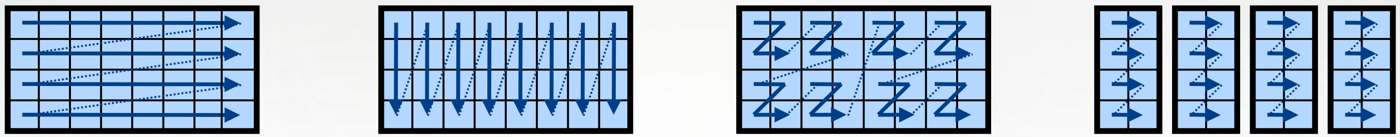


…?

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? …?

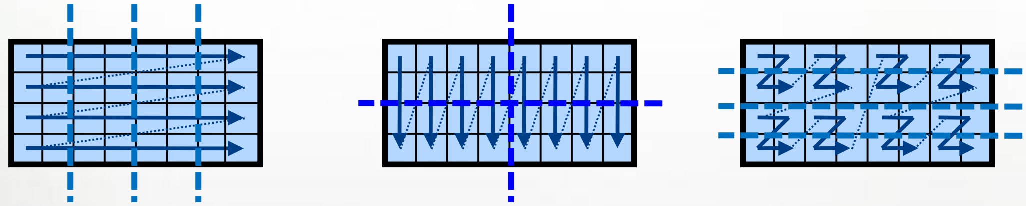
A: Chapel's *domain maps* are designed to give the user full control over such decisions

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 algorithm?



...?

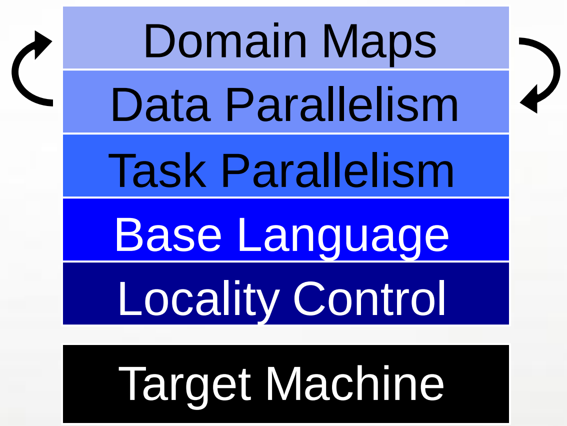
A: Chapel's domain maps are designed to give the user full control here, too

Multiresolution Design: Implementation

Multiresolution Design: Support multiple tiers of features

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

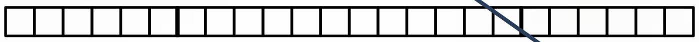
Chapel language concepts



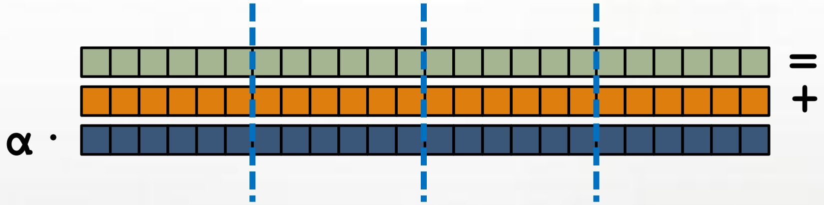
- build the higher-level concepts in terms of the lower
- permit the user to intermix layers arbitrarily

STREAM Triad: Chapel (multicore)

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



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

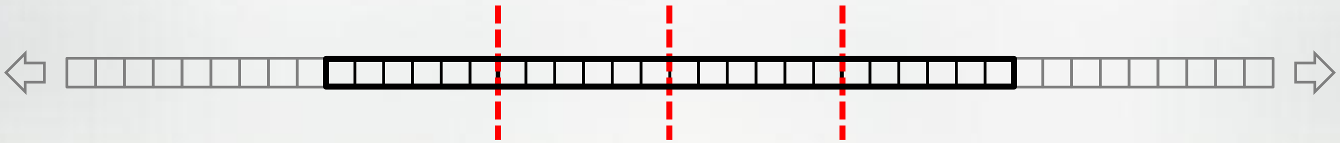



```
A = B + alpha * C;
```

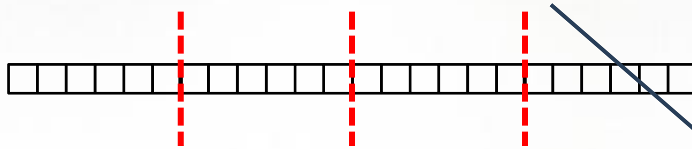
No domain map is specified => use the default one

- current locale owns all indices and values
- computation will execute using local processors only, in parallel

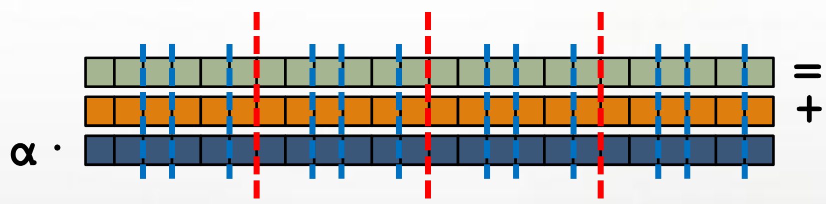
STREAM Triad: Chapel (multilocale, blocked)



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



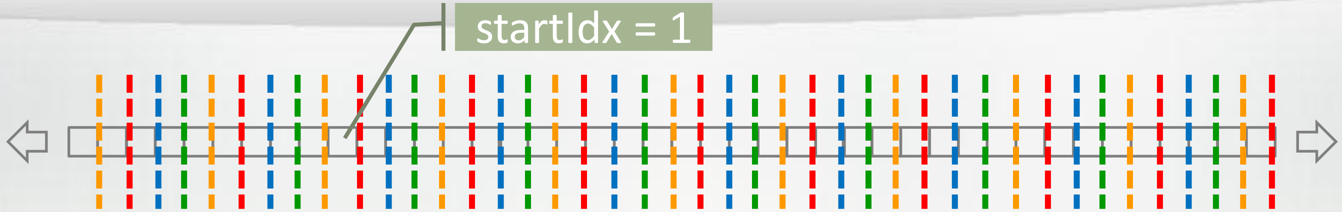
```
var A, B, C: [ProblemSpace] real;
```



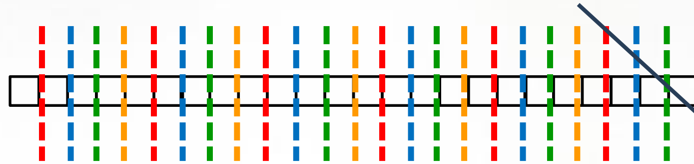

```
A = B + alpha * C;
```

Block domain map is chosen explicitly
- indices and values are distributed over all locales
- computation will execute on all locales and processors, in parallel

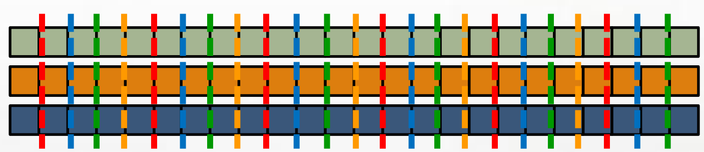
STREAM Triad: Chapel (multilocale, cyclic)




```
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

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 lower resolution for more control
- not required, but available when desired

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 author of `MyDomainMap` 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);
      }
  } }
}
```

For More Information on Domain Maps

HotPAR'10: *User-Defined Distributions and Layouts in Chapel: Philosophy and Framework*
Chamberlain, Deitz, Iten, Choi; June 2010

CUG 2011: *Authoring User-Defined Domain Maps in Chapel*
Chamberlain, Choi, Deitz, Iten, Litvinov; May 2011

Chapel release:

- Technical notes detailing domain map interface for programmers:
  $CHPL_HOME/doc/technotes/README.dsi
- Current domain maps:
  $CHPL_HOME/modules/dists/*.chpl
  layouts/*.chpl
  internal/Default*.chpl

For More Information on Leader-Follower Iterators

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

Chapel release:

- Primer example introducing leader-follower iterators:
  - examples/primers/leaderfollower.chpl
- Library of dynamic leader-follower range iterators:
  - *AdvancedIters* chapter of language specification

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
- ✓ Introducing Chapel
- ➢ Empirical Evaluation
- About the Project

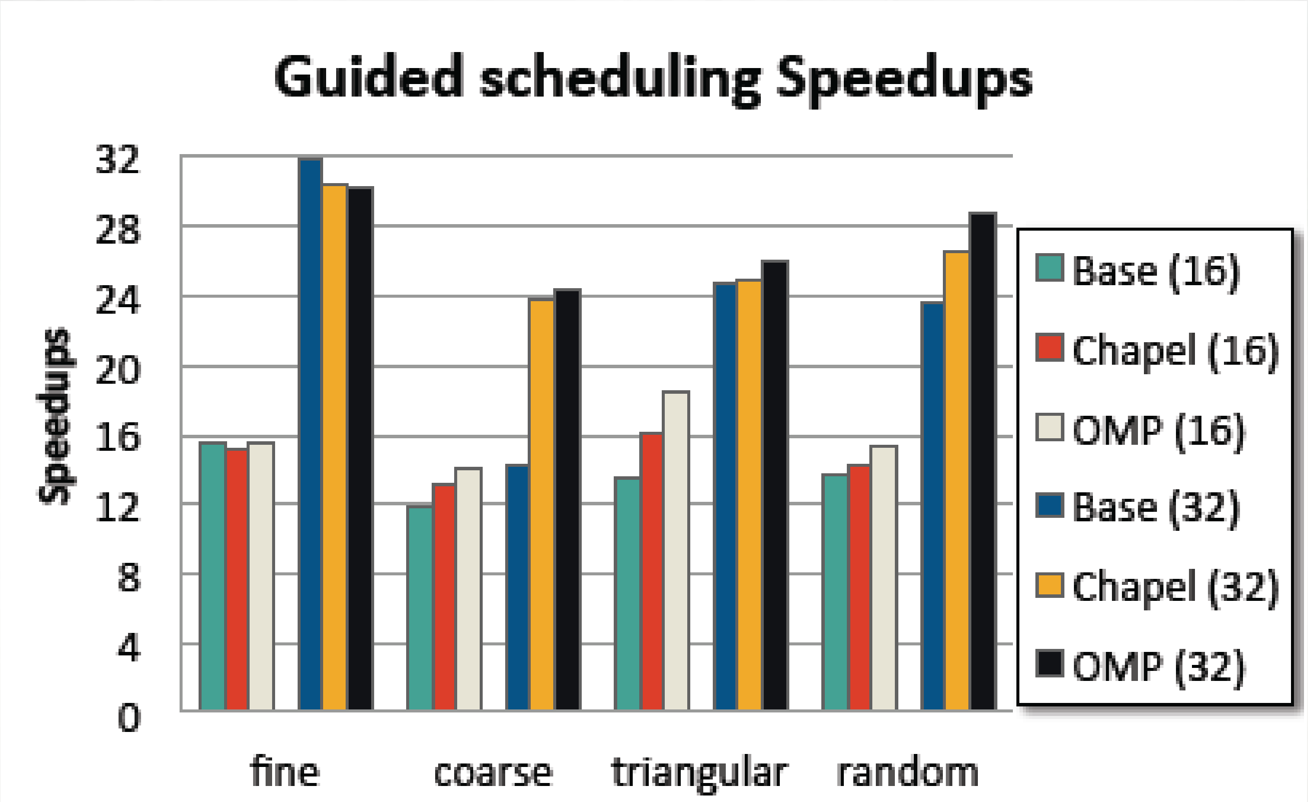
User-Defined Parallel Iterators

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



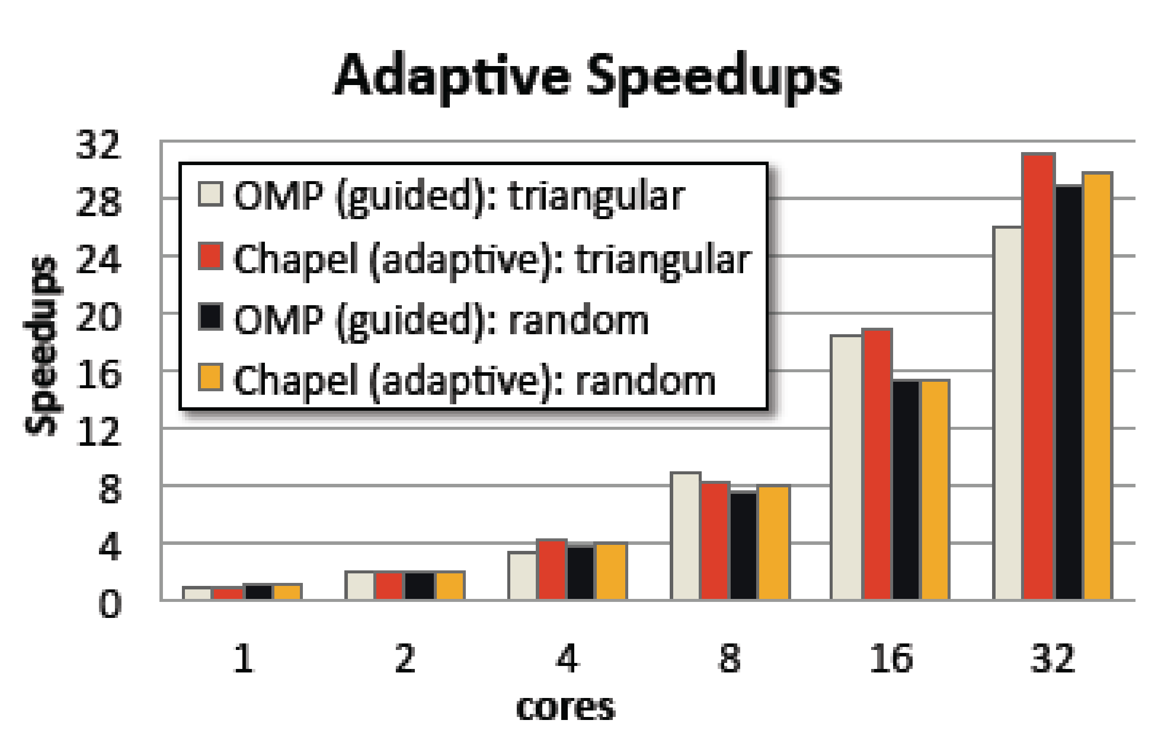
synthetic workloads:

- fine: 1 µsec × 1mln iters
- coarse: 0.1sec × 100 iters
- triangular: 100***i** µsec × **i**=1000...1
- random: 0.1*rand() sec × 1000 iters

speedup=1 => sequential C code

Base => Chapel's default iterator

Chapel Adaptive vs. OpenMP Guided



adaptive work stealing:

- scales better to more cores
  - distributed splitting reduces contention
- lowers the cost of splitting

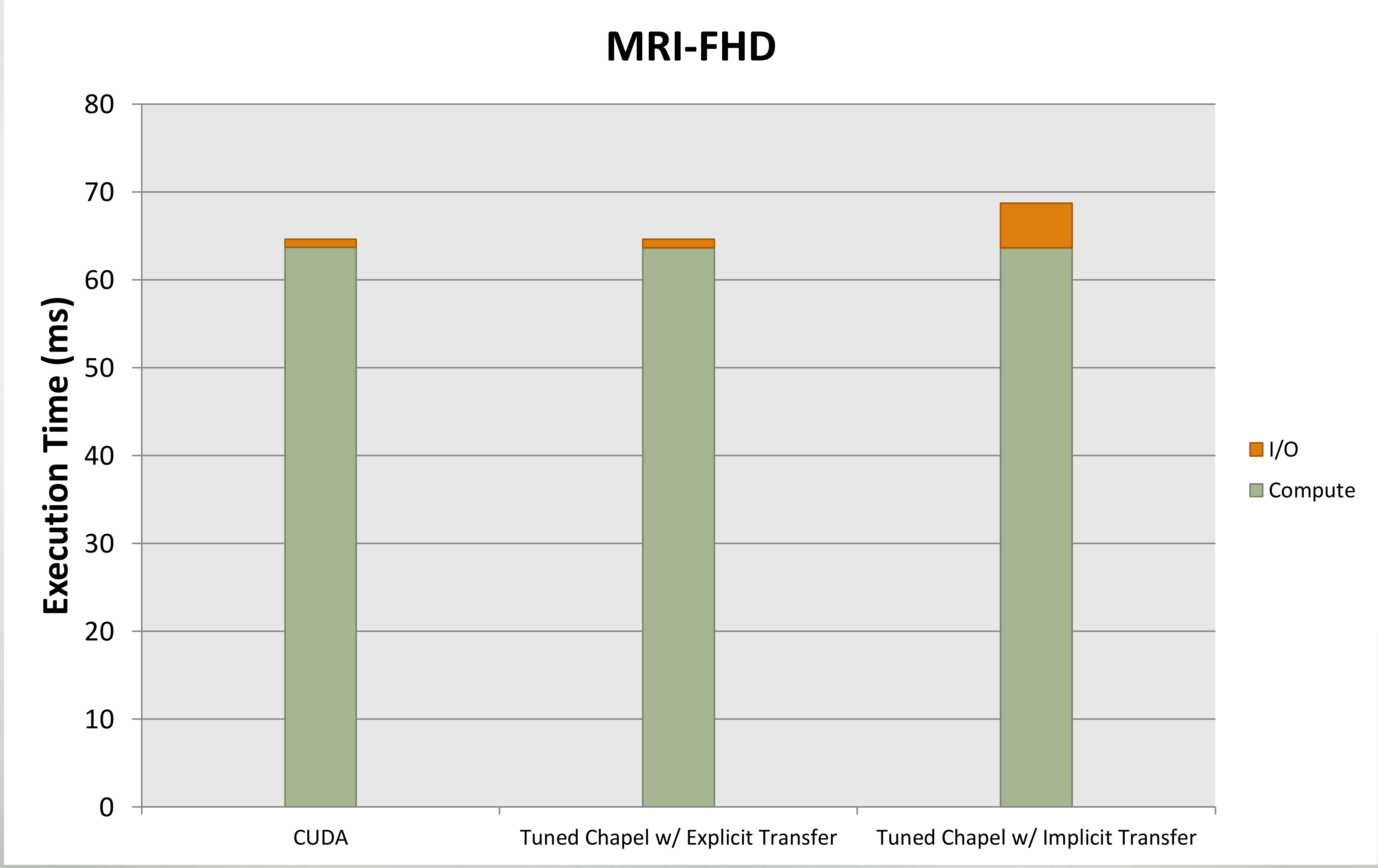
Targetting GPUs

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

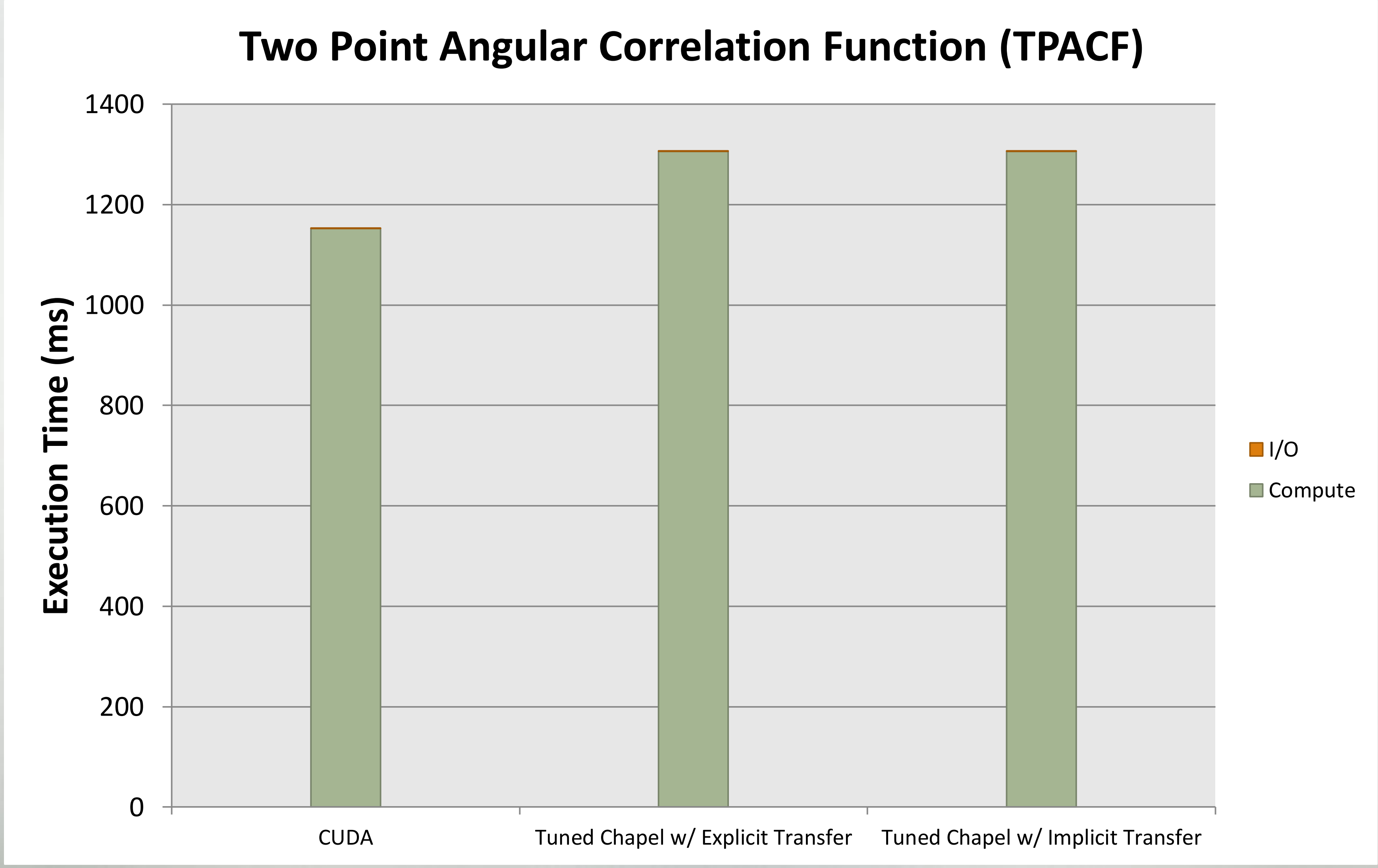
- Technology for running Chapel code on GPUs
  - implemented a domain map to place data and execute code on GPUs
  - added compiler support to emit CUDA code; additional optimizations
- Compared performance against hand-coded CUDA
  - competitive performance, less code

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

Aggregate Communication Optimization

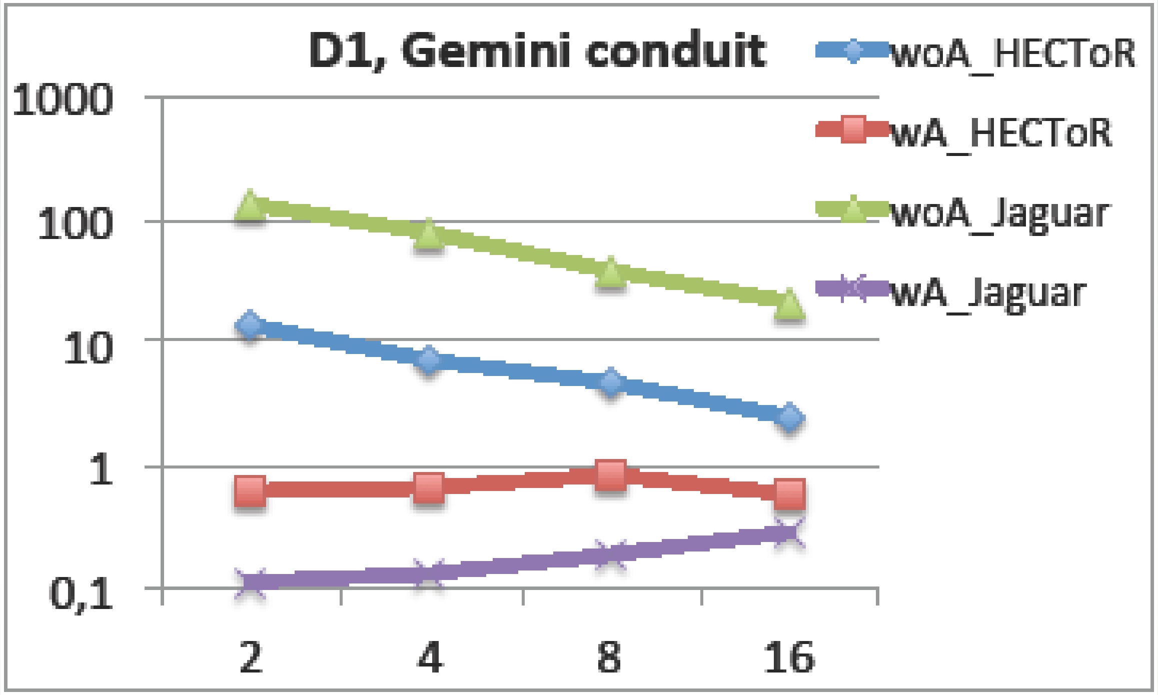
SBAC-PAD 2012: *Global Data Re-Allocation via Communication Aggregation,*
Sanz, Asenjo, López, Larrosa, Navarro, et al.; October 2012

- Reduced #messages for Chapel array assignments
  - from 1 per array element
  - to 1 per (source, destination) locale pair
  - for default, Block and Cyclic domain maps

Chapel's multi-resolution design allowed almost all of this optimization to be implemented at the Chapel source code level

- simple GASNet adaptors were written in C

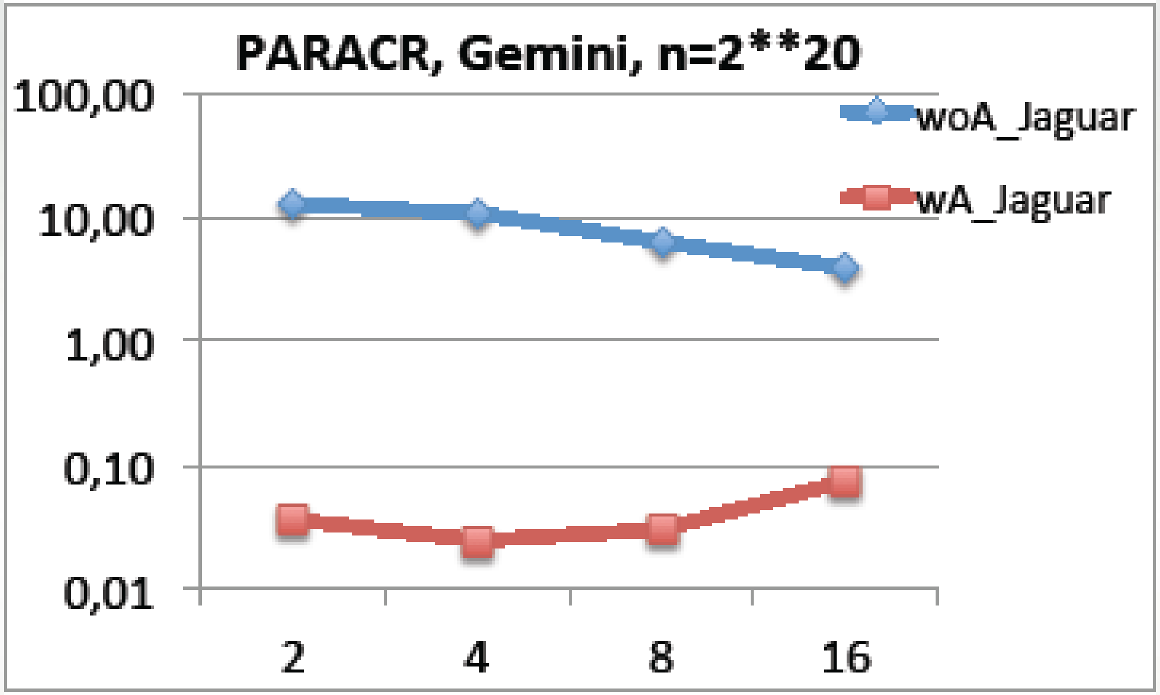
Array assignment, with/without aggregation



When #locales increases – #messages per locale...

- without aggregation => decreases
- with aggregation => increases
- asymptotically (constant #elements, large #locales) => similar #messages

PARACAR – Block-to-Cyclic redistribution



PARACAR is implemented in Chapel with arrays Block-distributed during earlier phases and Cyclic-distributed during later phases

Outline

- ✓ Motivation
- ✓ Multiresolution programming
- ✓ Introducing Chapel
- ✓ 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!

Implementation Status – Version 1.5 (April 2012)

In a nutshell:

- Most features work at a functional level
- Many performance optimizations remain
  - current team focus

This is a good time to:

- Try out the language and compiler
- Use Chapel for non-performance-critical projects
- Give us feedback to improve Chapel
- Use Chapel for parallel programming education

Some Next Steps

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

Collaborations (see chapel.cray.com/collaborations.html for details)

- **Bulk-Copy Optimization**
  - Rafael Asenjo et al. University of Málaga
  - reduce inter-node communication time
  - paper at SBAC-PAD, Oct 2012
- **Dynamic Iterators**
  - Angeles Navarro et al. University of Málaga
  - better dynamic and adaptive loop scheduling
  - paper at PGAS, Oct 2011
- **Parallel File I/O**
  - Rafael Larrosa et al. University of Málaga
  - speed up I/O of distributed arrays
  - paper at ParCo, Aug 2011
- **Tasking using Nanos++**
  - Alejandro Duran BSC/UPC
  - lightweight task switching

Collaborations, continued

- **Tasking using Qthreads:** Sandia (Rich Murphy, Kyle Wheeler, Dylan Stark)
  - **paper at CUG, May 2011**
- **Interoperability using Babel/BRAID:** LLNL (Tom Epperly, Adrian Prantl, et al.)
  - **paper at PGAS, Oct 2011**
- **Improved I/O & Data Channels:** LTS (Michael Ferguson)
- **LLVM back-end:** LTS (Michael Ferguson)
- **CPU-GPU Computing:** UIUC (David Padua, Albert Sidelnik, Maria Garzarán)
  - **paper at IPDPS, May 2012**
- **Interfaces/Generics/OOP:** CU Boulder (Jeremy Siek, Jonathan Turner)
- **Tuning/Portability/Enhancements:** ORNL (Matt Baker, Jeff Kuehn, Steve Poole)
- **Chapel-MPI Compatibility:** Argonne (Rusty Lusk, Pavan Balaji, Jim Dinan, et al.)

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/