Communication Optimization for the Chapel Programming Language

Michael Ferguson
Cray Inc.

October 23, 2015

Safe Harbor Statement

This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.

Talk Outline

- Introduce Chapel
- Why we need communication optimization
- Memory Consistency Models constrain optimization
- Sequential Consistency for Data Race Free Programs
- Optimizing communication with a cache for remote data
- Using LLVM to optimize communication

What is Chapel?

Chapel's Origins: HPCS

DARPA HPCS: High Productivity Computing Systems

- **Goal:** improve productivity by a factor of 10x
- **Timeframe:** summer 2002 – fall 2012
- Cray developed a new system architecture, network, software, …
  - this became the very successful Cray XC30™ Supercomputer Series

…and a new programming language: Chapel

Chapel Motivation

Q: Why doesn't parallel programming have an equivalent to Python / Matlab / Java / C++ / ___(your favorite programming language here)___ ?

- one that makes it easy to quickly get codes up and running
- one that is portable across system architectures and scales
- one that bridges the HPC, data analysis, and mainstream communities

A: We believe this is due not to any particular technical challenge, but rather a lack of sufficient…

…long-term efforts
…resources
…community will
…co-design between developers and users
…patience

Chapel is our attempt to change this

Chapel's Implementation

- **Being developed as open source at GitHub**
  - Licensed as Apache v2.0 software
- **Portable design and implementation, targeting:**
  - multicore desktops and laptops
  - commodity clusters and the cloud
  - HPC systems from Cray and other vendors
  - *in-progress:* manycore processors, CPU+accelerator hybrids, …

Chapel is a Collaborative, Community Effort

(and many others as well…)

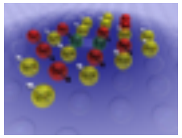
http://chapel.cray.com/collaborations.html

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 – ~20__: Cray ____; ~10,000,000 Processors
- TBD
- TBD: C/C++/Fortran + MPI + OpenMP/OpenACC/CUDA/OpenCL?

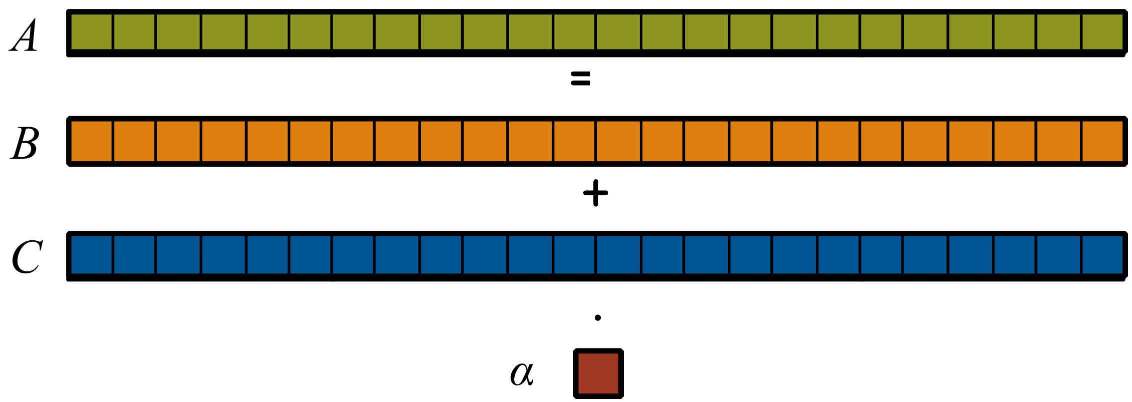
Or, perhaps something completely different?

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:

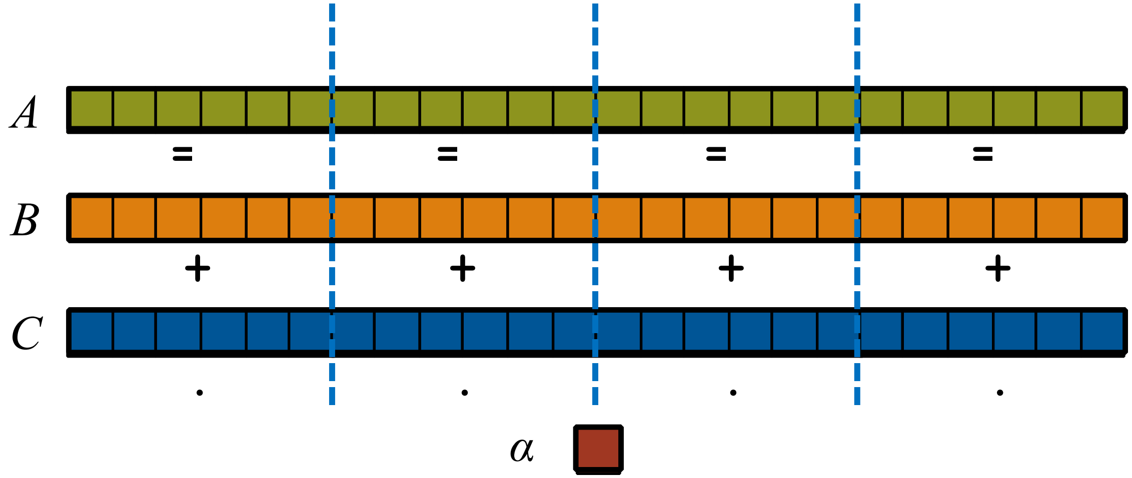


STREAM Triad: a trivial parallel computation

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:

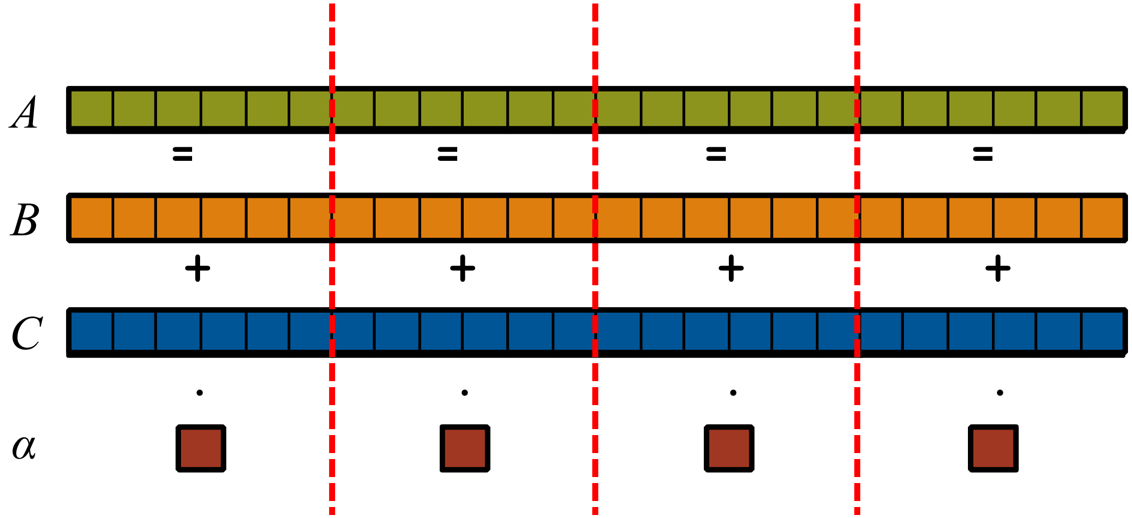


STREAM Triad: a trivial parallel computation

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):

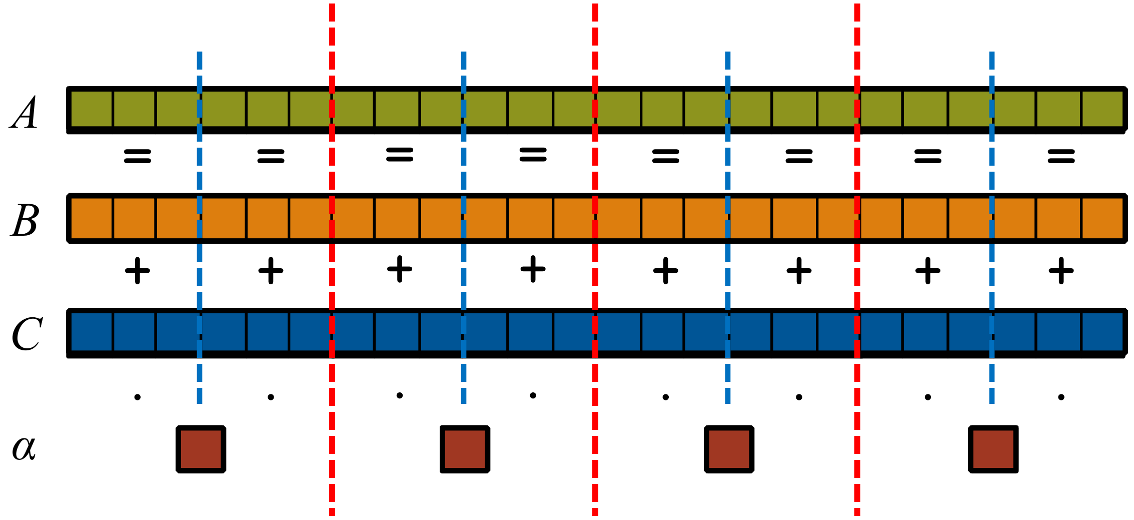


STREAM Triad: a trivial parallel computation

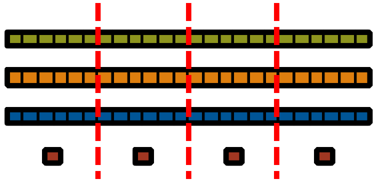
Given: m-element vectors A, B, C

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

In pictures, in parallel (distributed memory multicore):



STREAM Triad: MPI

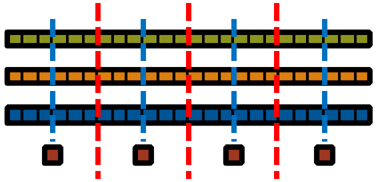


MPI

```
#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] = 1.0;
}

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);
```

STREAM Triad: MPI+OpenMP



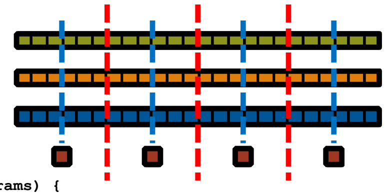
MPI + OpenMP

```
#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] = 1.0;
  }

  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);
```

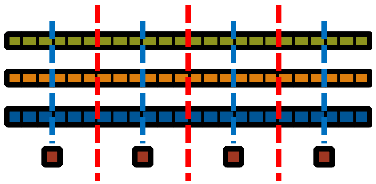
STREAM Triad: MPI+OpenMP vs. CUDA

MPI + OpenMP



```
#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] = 1.0;
  }

  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, .5f, N);
  set_array<<<dimGrid,dimBlock>>>(d_c, .5f, 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];
}
```

HPC suffers from too many distinct notations for expressing parallelism and locality

Why so many programming models?

HPC has traditionally given users…

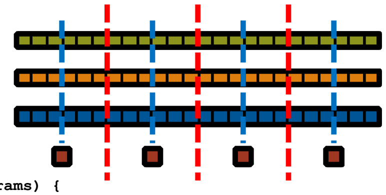
…low-level, *control-centric* programming models
…ones that are closely tied to the underlying hardware
…ones that support only a single type of parallelism

Type of HW Parallelism	Programming Model	Unit of Parallelism
Inter-node	*MPI*	*executable*
Intra-node/multicore	*OpenMP / pthreads*	*iteration/task*
Instruction-level vectors/threads	*pragmas*	*iteration*
GPU/accelerator	*Open[MP|CL|ACC] / CUDA*	*SIMD function/task*

benefits: lots of control; decent generality; easy to implement
downsides: lots of user-managed detail; brittle to changes

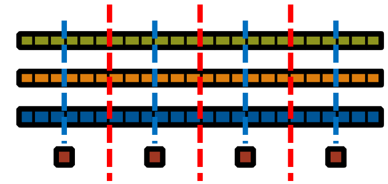
Rewinding a few slides…

MPI + OpenMP



```
#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) {

  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] = 1.0;
  }

  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 );

  set_array<<<dimGrid,dimBlock>>>(d_b, .5f, N);
  set_array<<<dimGrid,dimBlock>>>(d_c, .5f, 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];
}
```

HPC suffers from too many distinct notations for expressing parallelism and locality

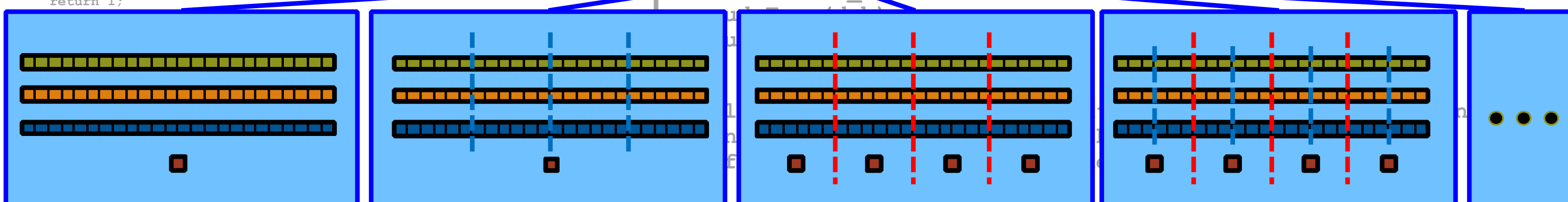
STREAM Triad: Chapel

MPI + OpenMP

Chapel

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

const ProblemSpace = {1..m} dmapped …;

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

B = 2.0;
C = 1.0;

A = B + alpha * C;
```

the special sauce



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

Motivating Chapel Themes

1) General Parallel Programming
2) Global-View Abstractions
3) Multiresolution Design
4) Control over Locality/Affinity
5) Reduce HPC ↔ Mainstream Language Gap

Motivating Chapel Themes

1) General Parallel Programming
2) Global-View Abstractions
3) Multiresolution Design
4) PGAS: Control over Locality/Affinity
5) Reduce HPC ↔ Mainstream Language Gap

1) General Parallel Programming

With a unified set of concepts...

...express any parallelism desired in a user's program

- **Styles:** data-parallel, task-parallel, concurrency, nested, …
- **Levels:** model, function, loop, statement, expression

...target any parallelism available in the hardware

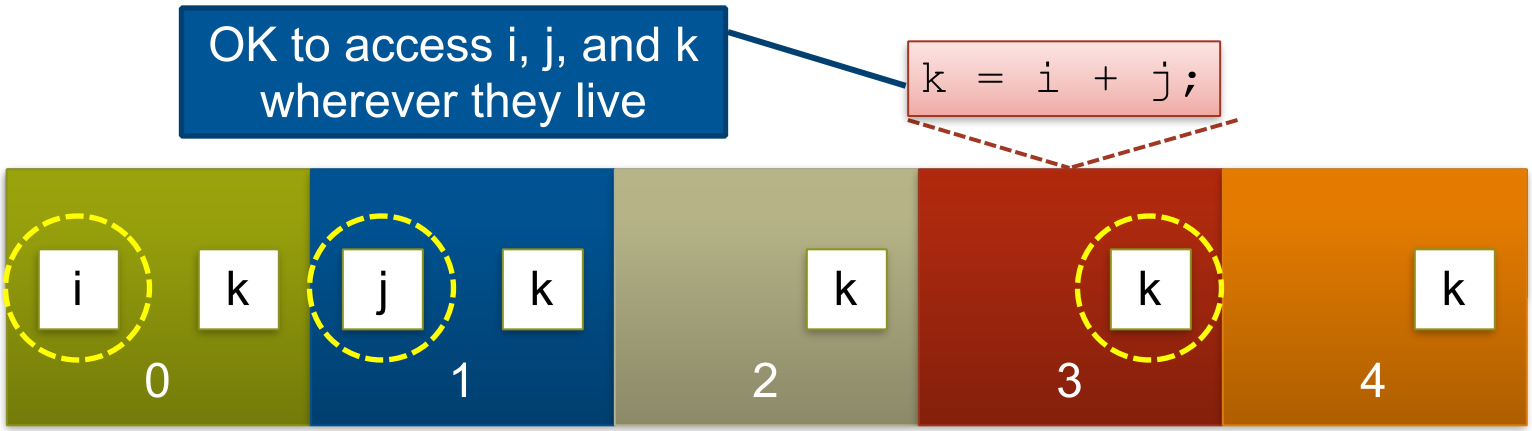
- **Types:** machines, nodes, cores, instruction

Type of HW Parallelism	Programming Model	Unit of Parallelism
Inter-node	*Chapel*	*task(or executable)*
Intra-node/multicore	*Chapel*	*iteration/task*
Instruction-level vectors/threads	*Chapel*	*iteration*
GPU/accelerator	*Chapel*	*SIMD function/task*

PGAS Programming in a Nutshell

Global Address Space:

- permit parallel tasks to access variables by naming them
  - regardless of whether they are local or remote
  - compiler / library / runtime will take care of communication



Images / Threads / Locales / Places / etc. (think: “compute nodes”)

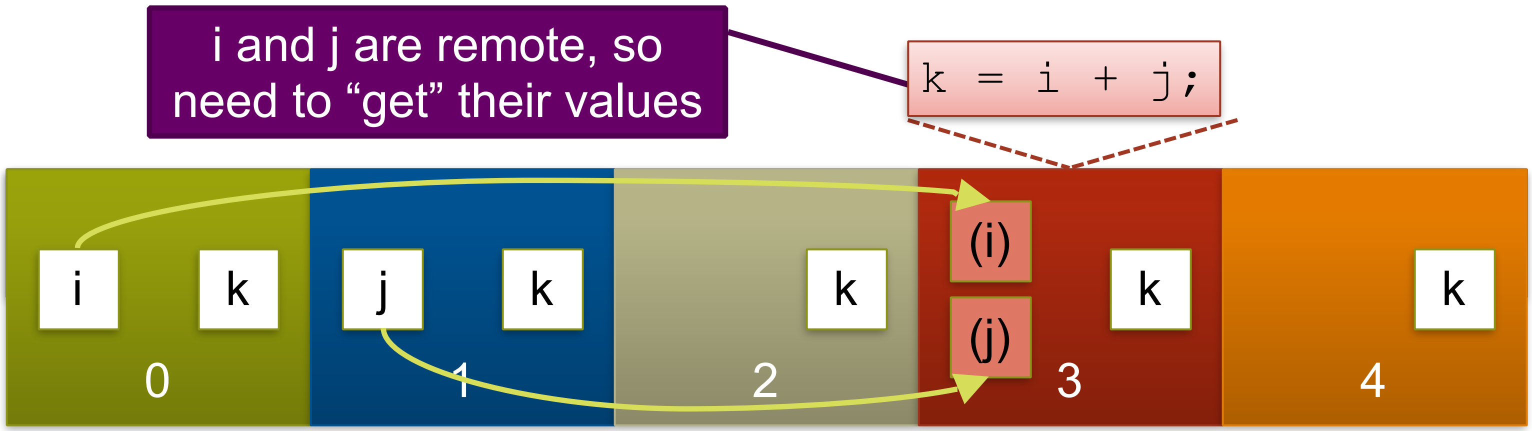
PGAS Programming in a Nutshell

Global Address Space:

- permit parallel tasks to access variables by naming them
  - regardless of whether they are local or remote
  - compiler / library / runtime will take care of communication

Partitioned:

- establish a strong model for reasoning about locality
  - every variable has a well-defined location in the system
  - local variables are typically cheaper to access than remote ones



Images / Threads / Locales / Places / etc. (think: "compute nodes")

PGAS Programming in a Nutshell

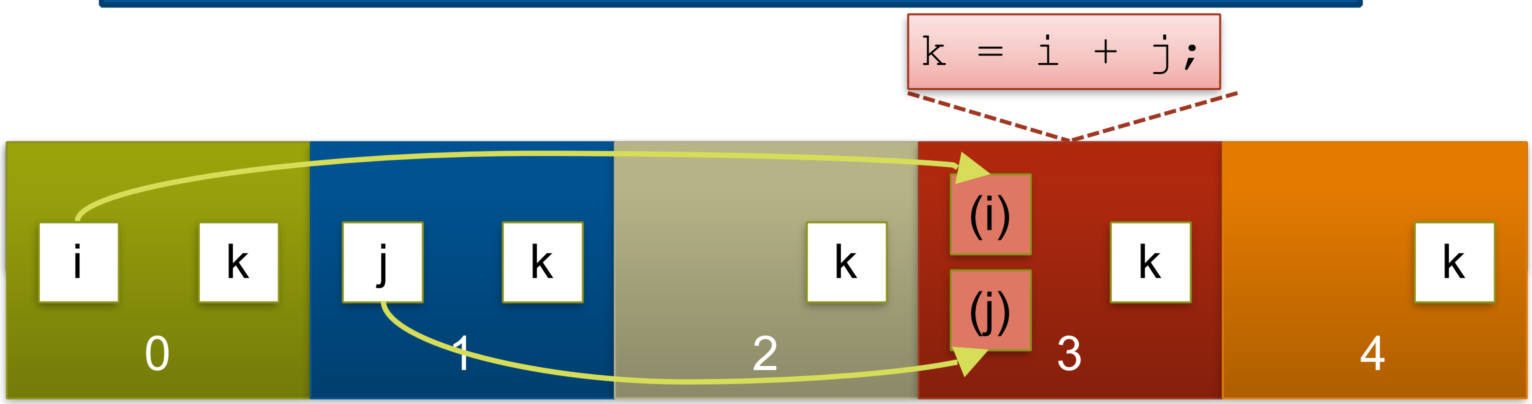
Global Address Space:

- permit parallel tasks to access variables by naming them
  - regardless of whether they are local or remote
  - compiler / library / runtime will take care of communication

Partitioned:

- estal
  - ev
  - loc es

Communication is implicit!
One sided GET and PUT.

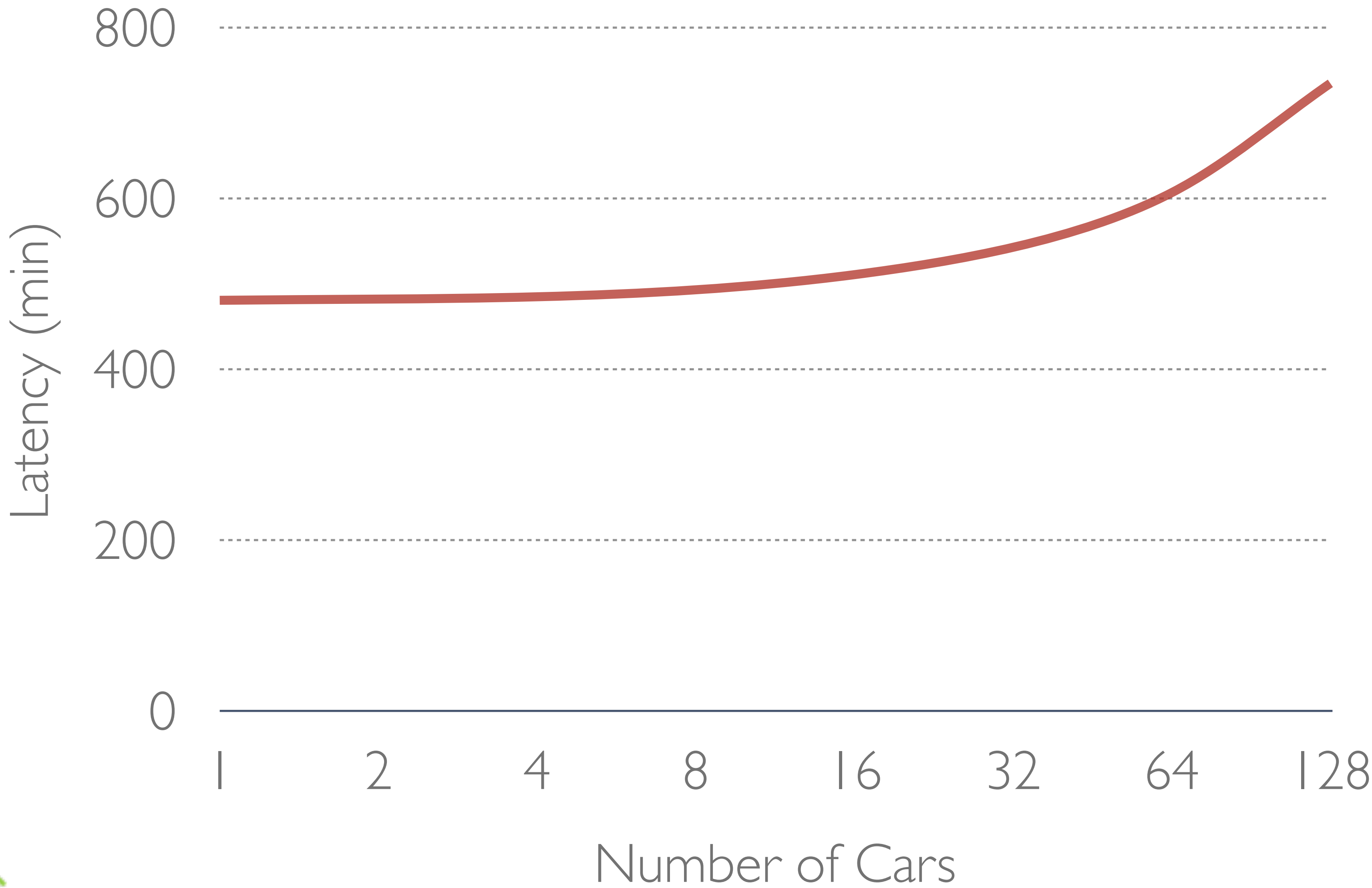


Images / Threads / Locales / Places / etc. (think: “compute nodes”)

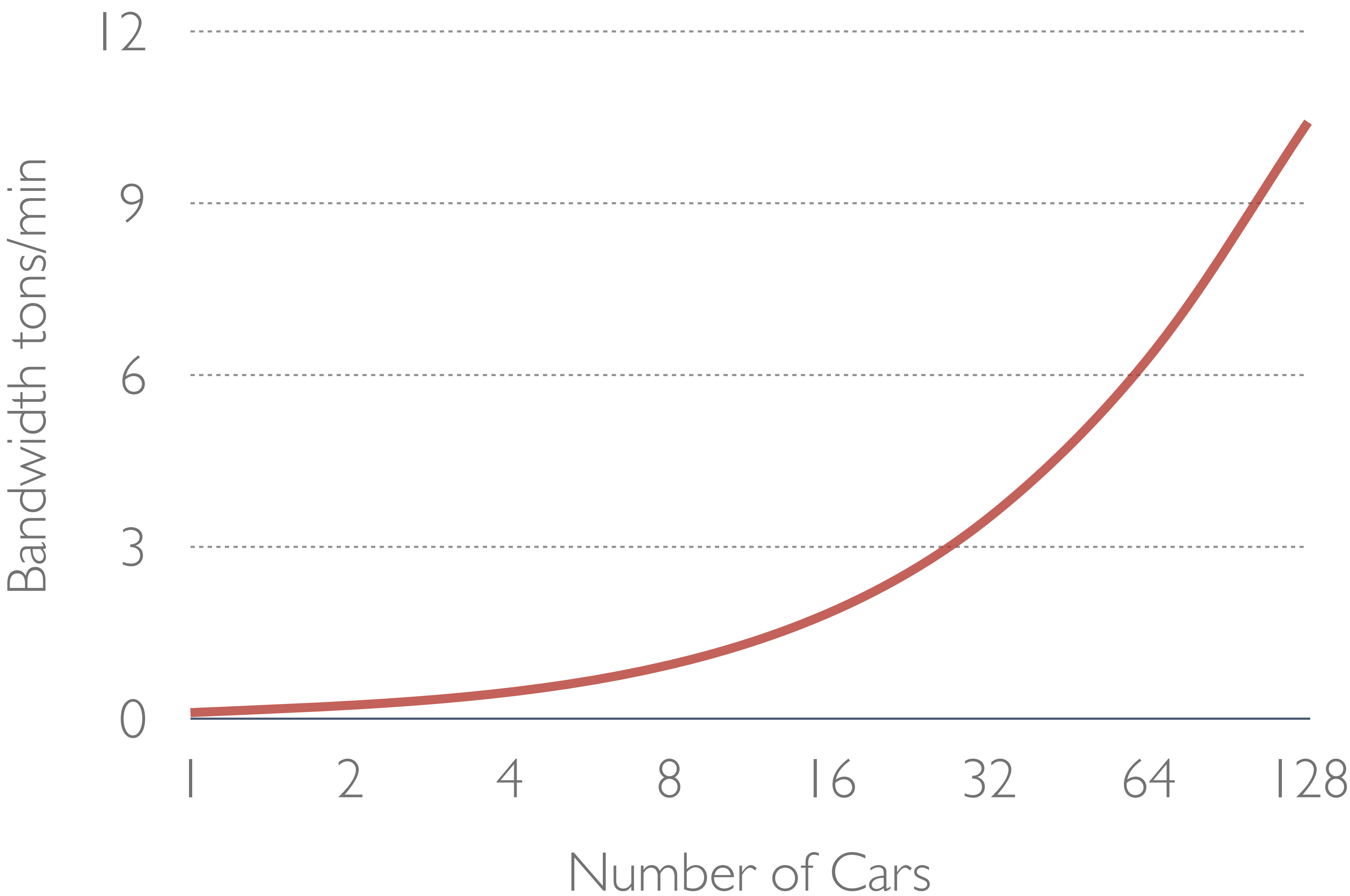
WHY COMMUNICATION OPTIMIZATION?

CC Flickr/ajmexico

TRAIN LATENCY (8 HOUR TRIP, 60 TON CARS, 60 SEC/CAR)

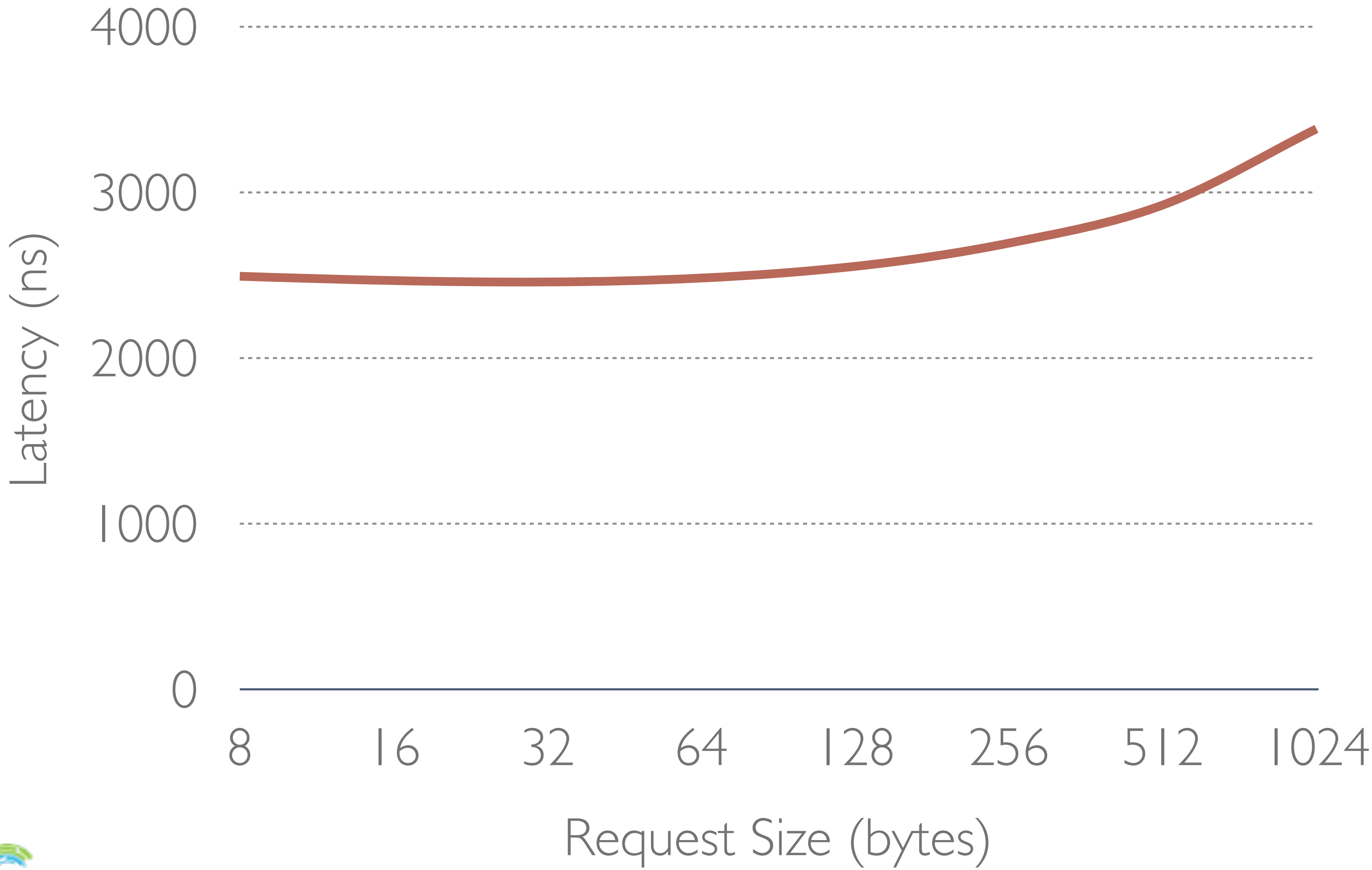


TRAIN BANDWIDTH

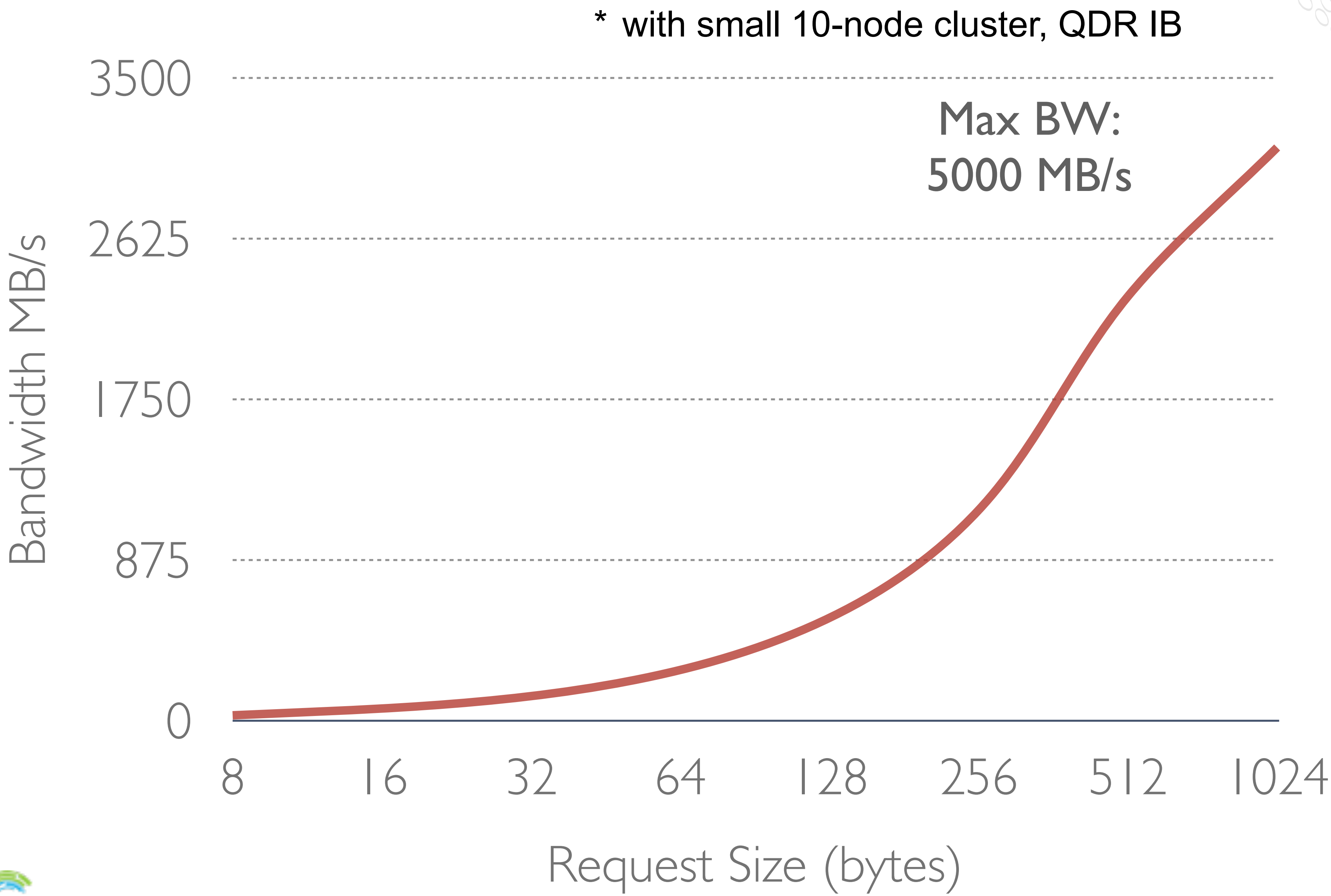


INFINIBAND (IB) LATENCY

* with small 10-node cluster, QDR IB



INFINIBAND (IB) BANDWIDTH



MEMORY MODELS CONSTRAIN PREFETCH AND WRITE-BEHIND

Library of Congress

AGGREGATION

CC Flickr/ajmexico

OVERLAP

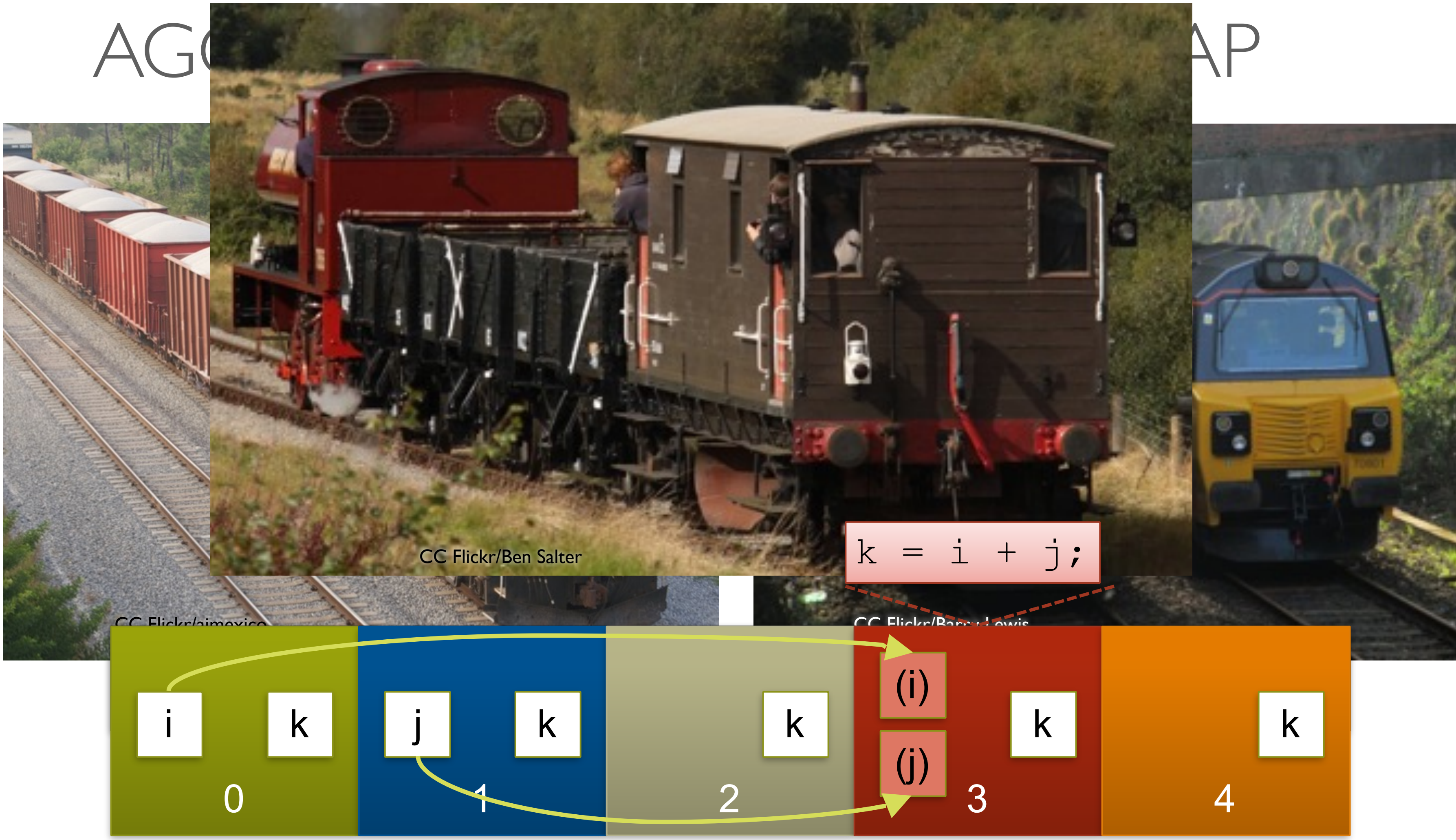

CC Flickr/Barry Lewis

AG AP

CC Flickr/ajmexico

CC Flickr/Ben Salter

CC Flickr/Barry Lewis


AG
AP
CC Flickr/Ben Salter
k = i + j;
CC Flickr/aimexico
i
k
j
k
k
(i)
(j)
k
k
0
1
2
3
4

A RACY PROGRAM

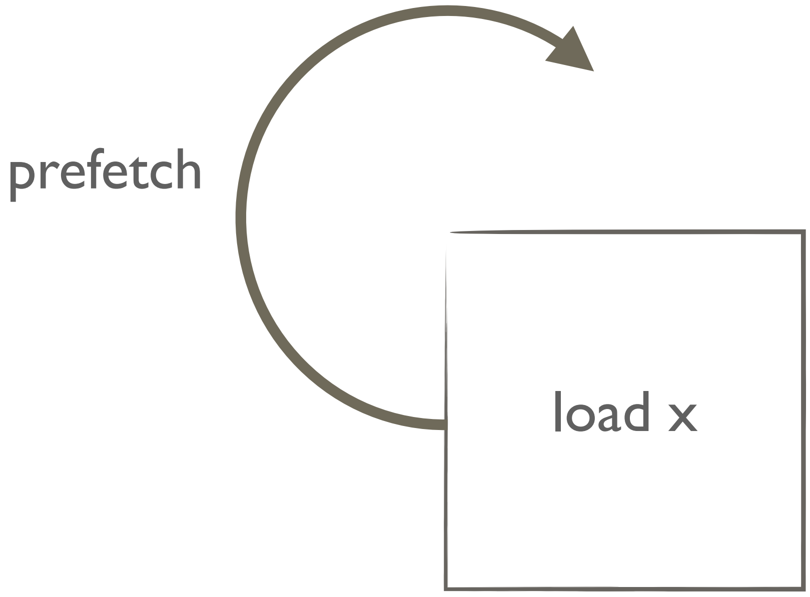
Thread 1	Thread 2
x = 42;	while 0 == notify { /* wait */ }
notify = 1;	compute_with(x);

A RACY PROGRAM

Thread 1	Thread 2
x = 42;	while 0 == notify { /* wait */ }
notify = 1;	compute_with(x);

compiler *or* processor

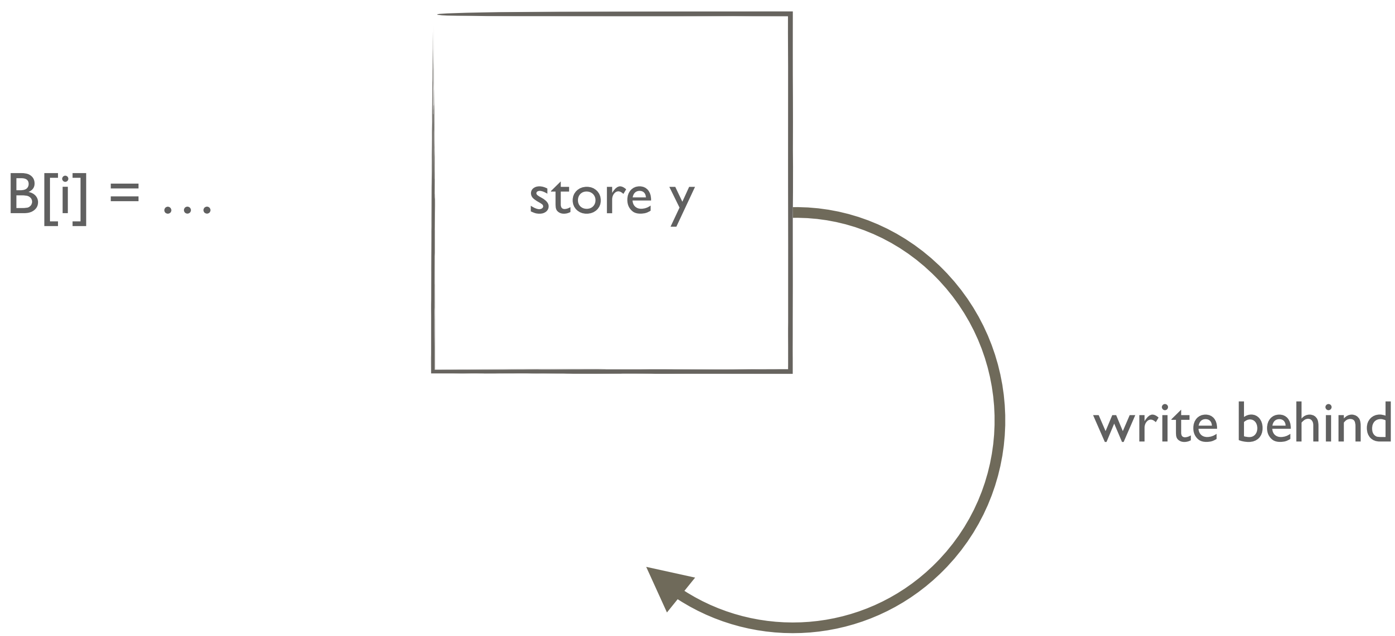
Thread 1	Thread 2
r1 = 42;	r2 = notify; while 0 == r2 { /* wait */ }
notify = 1; x = r1;	compute_with(x);



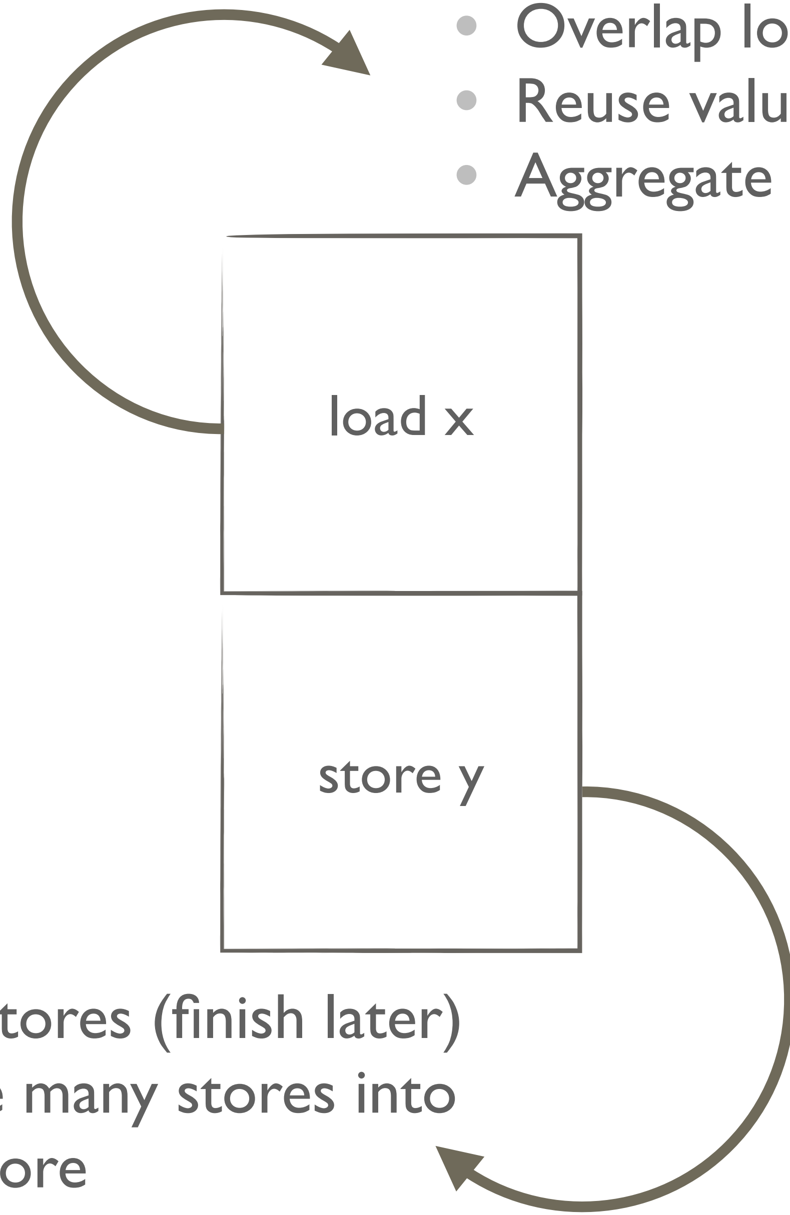
… = A[i]

Compiler *and* processor would like to start loads earlier in order to hide memory latency. We'll call that *prefetch*.

Compiler *and* processor would like to complete stores later in order to hide memory latency. We'll call that *write behind*.



- Overlap loads (start early)
- Reuse values from earlier load
- Aggregate loads (cache lines)



- Overlap stores (finish later)
- Aggregate many stores into a single store

PGAS Programming in a Nutshell

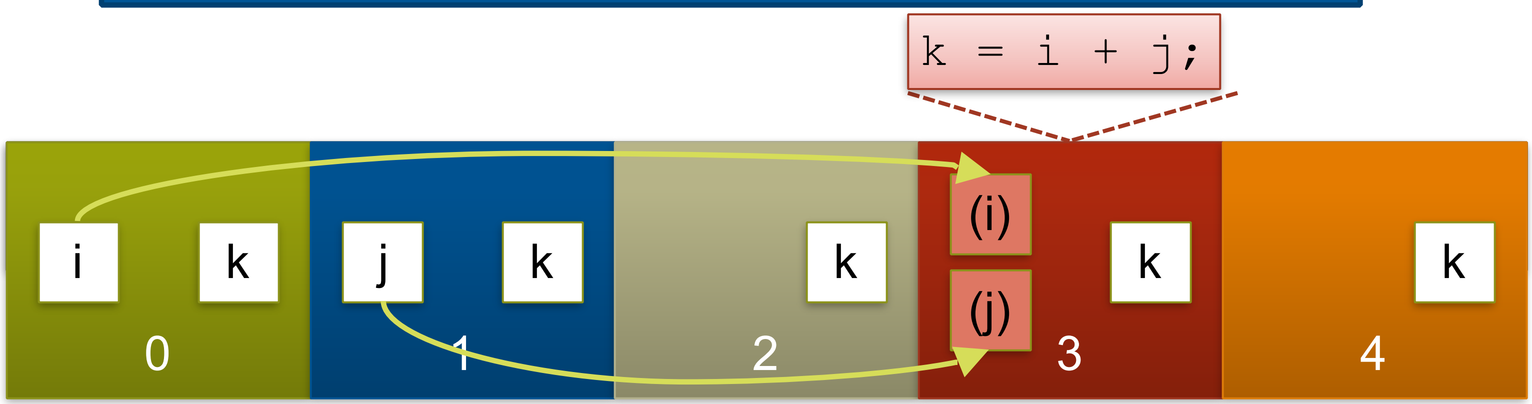
Global Address Space:

- permit parallel tasks to access variables by naming them
  - regardless of whether they are local or remote
  - compiler / library / runtime will take care of communication

Partitioned:

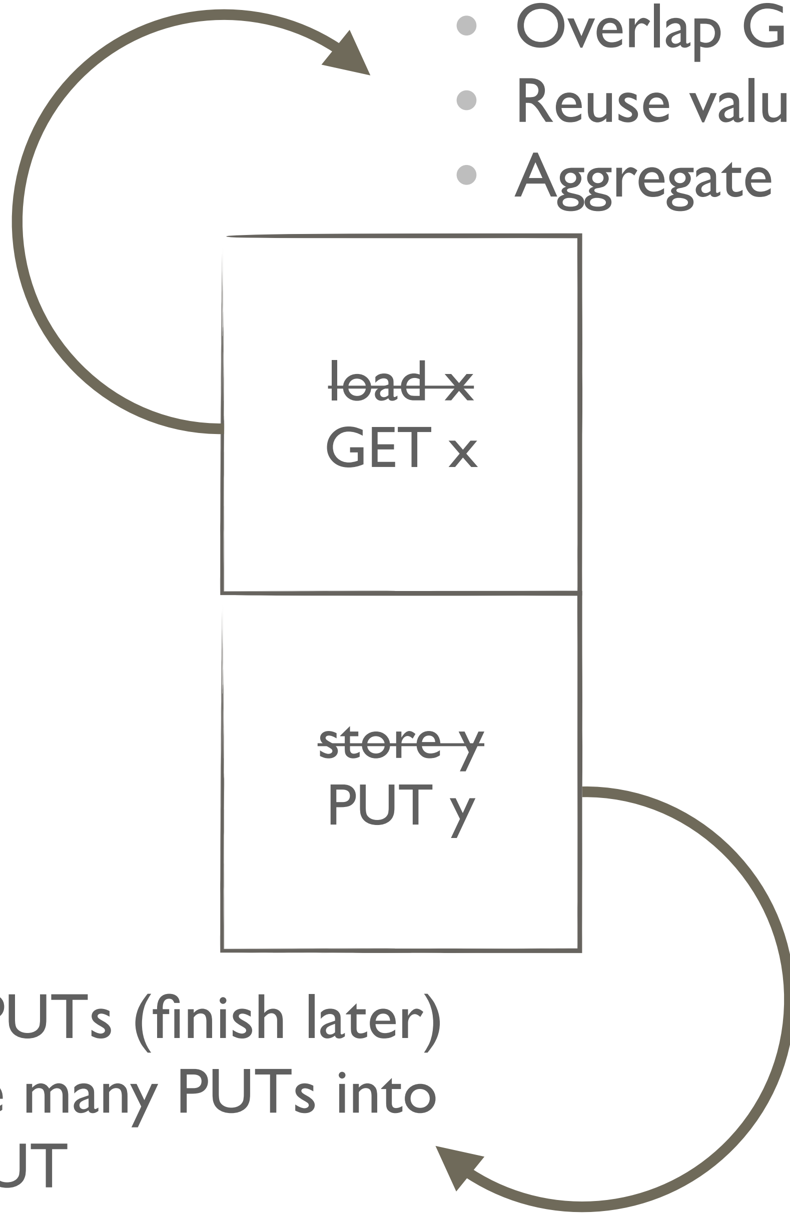
- estal
  - ev
  - loc es

Communication is implicit!
One sided GET and PUT.



Images / Threads / Locales / Places / etc. (think: "compute nodes")

- Overlap GETs (start early)
- Reuse values from earlier GET
- Aggregate GETs (cache lines)



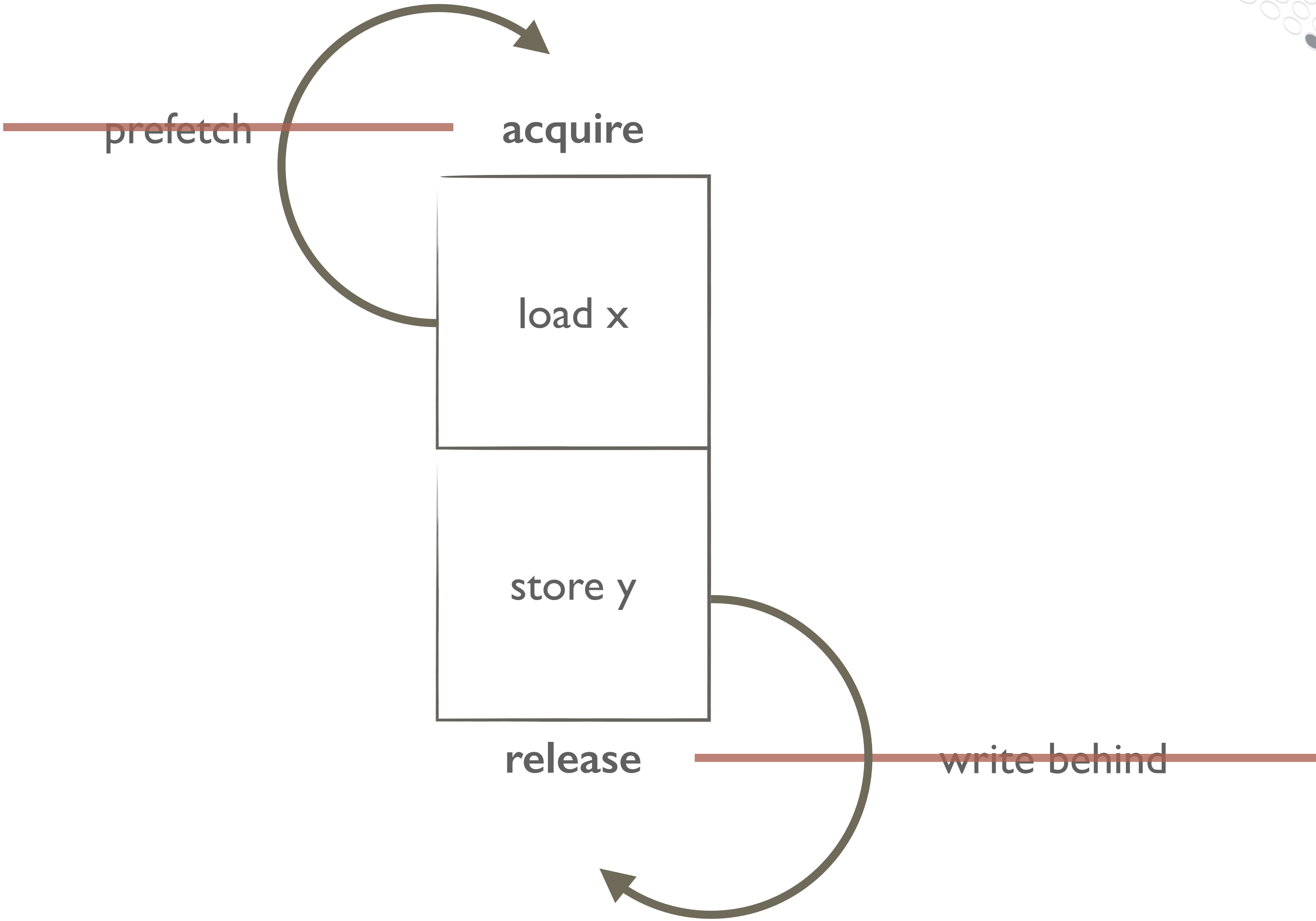
- Overlap PUTs (finish later)
- Aggregate many PUTs into a single PUT

REMEMBER THE RACY PROGRAM?

Thread 1	Thread 2
x = 42;	while 0 == notify { /* wait */ }
notify = true;	compute_with(x);

compiler *or* processor

Thread 1	Thread 2
r1 = 42;	r2 = notify; while 0 == r2 { /* wait */ }
notify = 1; x = r1;	compute_with(x);


~~prefetch~~
acquire
load x
store y
release
~~write behind~~

~~prefetch~~

acquire

load x

store y

release

~~write behind~~

atomic, sync
provide both
acquire, release

acquire lock
release lock

UNDERSTANDING SC-DRF

- atomic operations in Chapel and C++ support:
  - *memory_order_relaxed* "atomic only"
  - *memory_order_acquire* "acquire"
  - *memory_order_release* "release"
  - *memory_order_seq_cst* "sequentially consistent"
- Beware! No global total order for relaxed, acquire, and release. Instead, the order is per atomic variable.
- Data race free programs using only *memory_order_seq_cst* are easier to reason about: there is a global total order on memory operations.
- So *memory_order_seq_cst* is the default

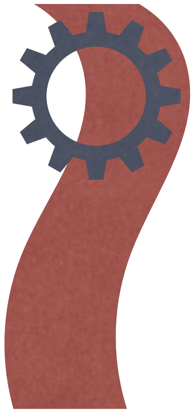

CASTLE ROCK.

SC FOR DRF

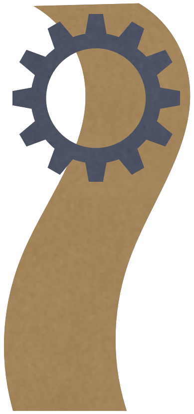
Memory model for `C11`, `C++11`, `Chapel`: *data race free programs are sequentially consistent*

- See Adve, S.V., Boehm, H.-J. 2010. Memory models: a case for rethinking parallel languages and hardware. Communications of the ACM 53(8): 90–101. http://cacm.acm.org/magazines/2010/8/96610-memory-models-a-case-for-rethinking-parallel-languages-and-hardware/fulltext
- Chapel has a new specification chapter describing the memory consistency model. See http://chapel.cray.com/spec/spec-0.98.pdf section 29, page 217.

Program Order

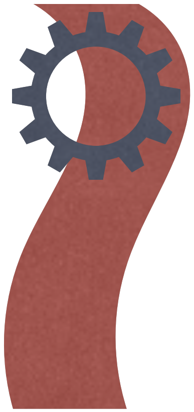


```
x = 1;
```

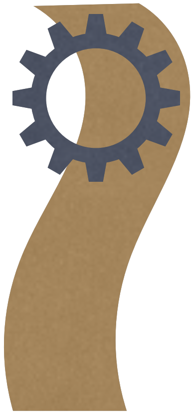


Memory Order

Program Order



```
x = 1;
```



Memory Order

`x = 1;`

Program Order

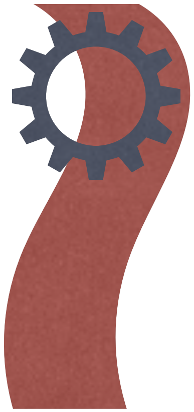
```
x = 1;
```

```
y = 3;
z = 4;
```

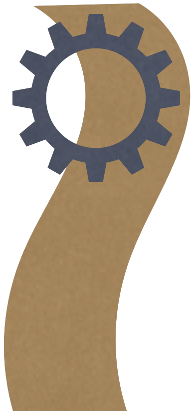
Memory Order

`x = 1;`

Program Order



```
x = 1;
```

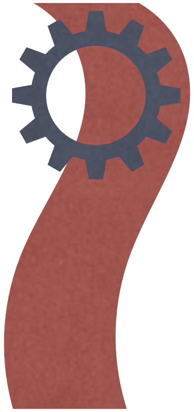


```
y = 3;
z = 4;
```

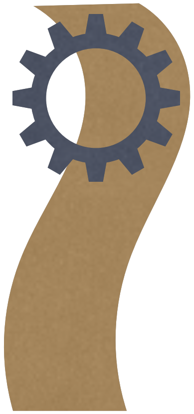
Memory Order

x = 1;
y = 3;
z = 4;

Program Order



```
x = 1;
```



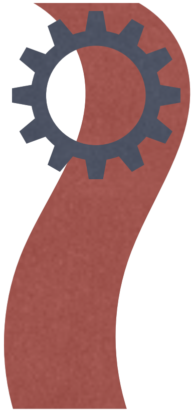
```
y = 3;
z = 4;
```

Some re-orderings are allowed.

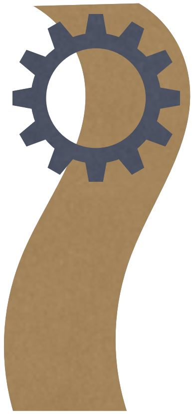
Memory Order

x = 1;
z = 4;
y = 3;

Program Order



```
x = 1;
x = 2;
a = x;
```

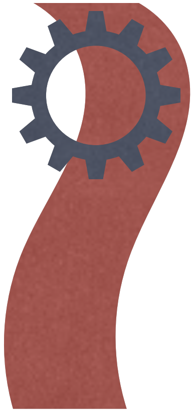


```
y = 3;
z = 4;
b = y;
```

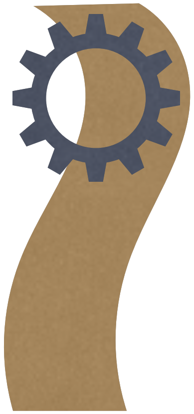
Memory Order

x = 1;
z = 4;
y = 3;
x = 2;
b = y;
a = x;

Program Order



```
x = 1;
x = 2;
a = x;
```

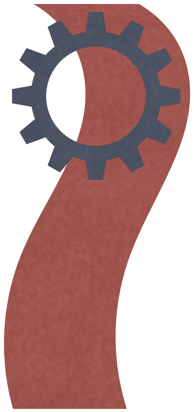


```
y = 3;
z = 4;
b = y;
```

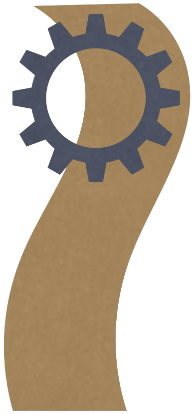
Memory Order

x = 1;
x = 2;
a = x;
y = 3;
z = 4;
b = y;

Program Order



```
x = 1;
x = 2;
a = x;
```

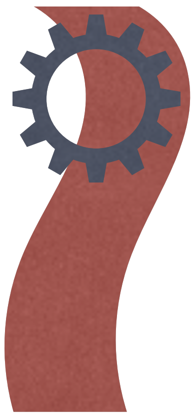


```
y = 3;
y = 4;
b = y;
```

Memory Order

x = 1;
y = 3;
x = 2;
y = 4;
b = y;
a = x;

Program Order



```
x = 1;
x = 2;
a = x;
```

Memory Order

```
x = 2;
x = 1;
a = x;
```

Not legal!

Sequential programs must work as if executed in program order

WEAKER MEMORY CONSISTENCY?

```
1 x starts at 0;
  ...
  if someOption then
2   x = 2;
  if someOtherOption then
3   x = 3;
4 return x;
```

WEAK MEMORY CONSISTENCY?

```
1 x starts at 0;
    ...
    ...
2     PUT 2 into x;
    ...
3     PUT 3 into x;
4 GET x;
```

Chapel

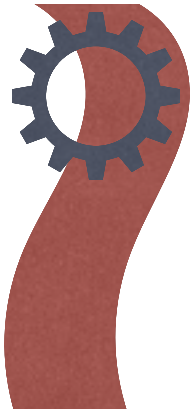
result must be 3

OpenSHMEM

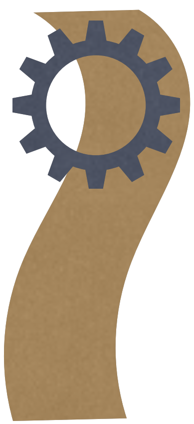
result could be 0, 2, or 3

Program Order

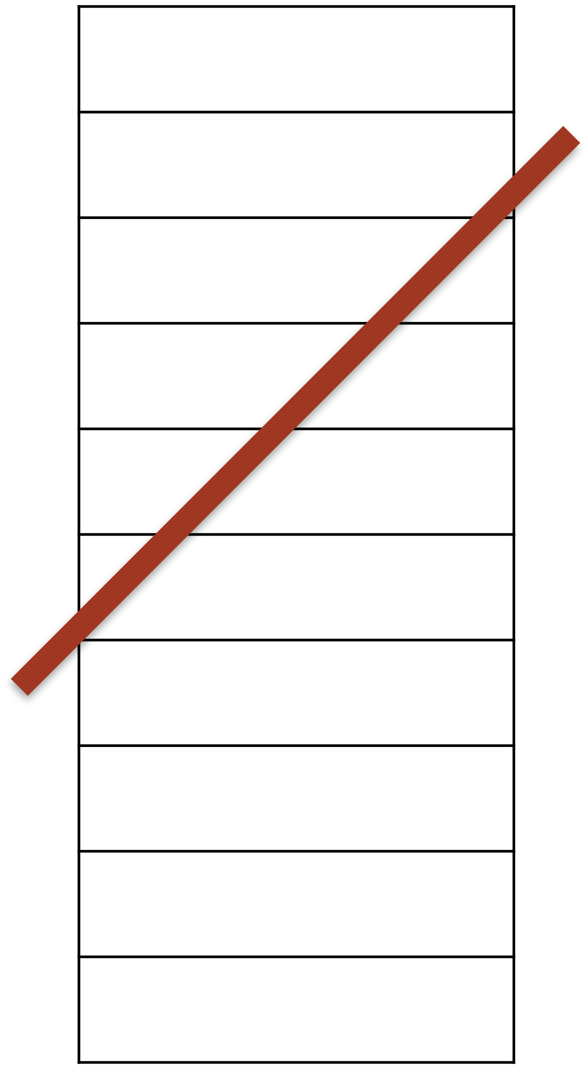
Memory Order



```
x = 1;
ok= 1;
```



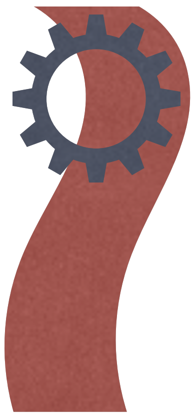
```
b =ok;
c = x;
```



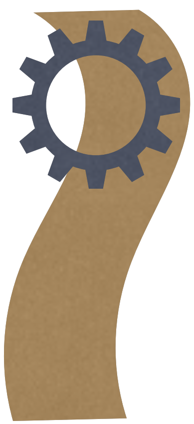
Bad program: Data Race. No global order! This outcome is possible:

```
b == 1
c == 0
```

Program Order



```
x = 0x1234;
```

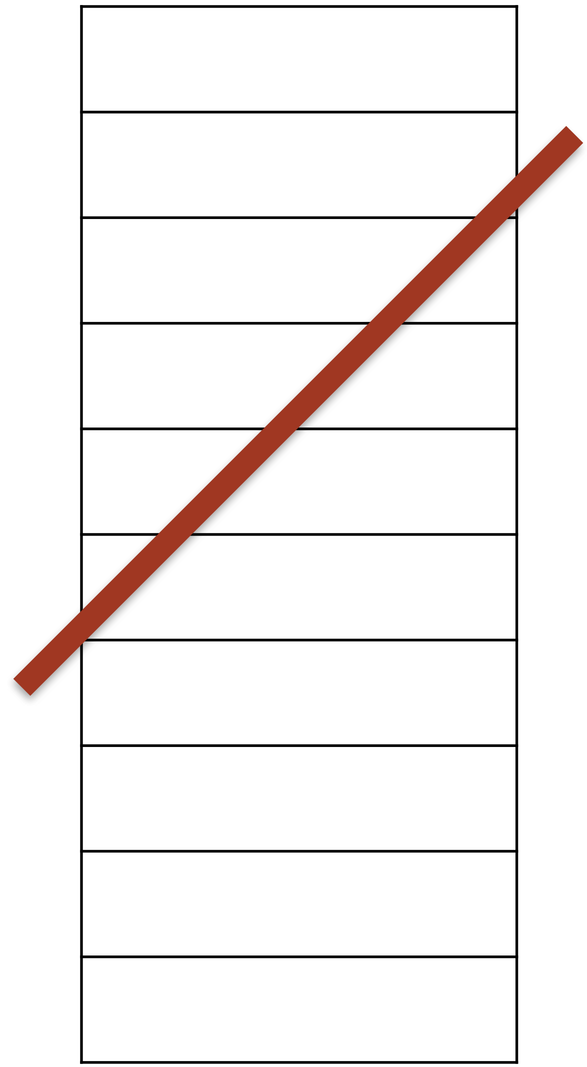


```
x = 0xABCD;
c = x;
```

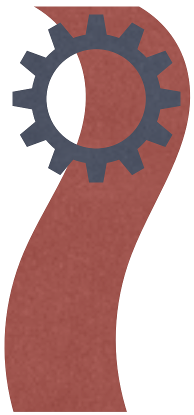
Bad program: Data Race. No global order! This outcome is possible:

```
c == 0xAB34
```

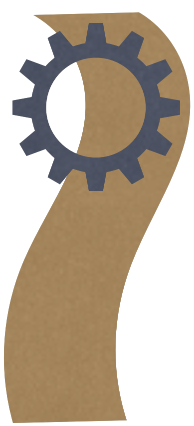
Memory Order



Program Order



```
x = 1;
ok= 1;
```

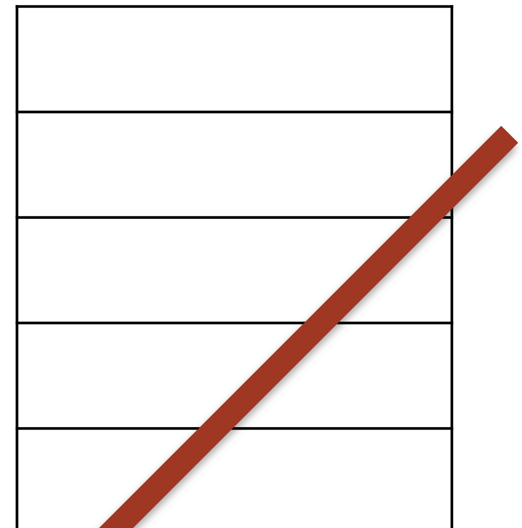


```
b =ok;
c = x;
```

Bad program: Data Race. No global order! This outcome is possible:

```
b == 1
c == 0
```

Memory Order



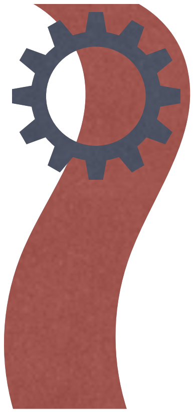
write behind could reorder:

```
ok= 1;
x = 1;
```

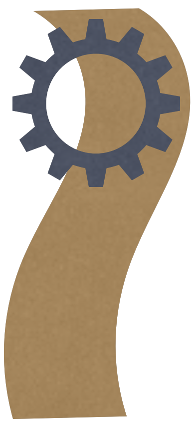
read ahead could reorder:

```
c = x;
b =ok;
```

Program Order



```
atomic = 1;
c = atomic;
```



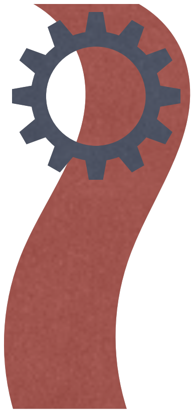
```
b = atomic;
atomic = 2;
```

In Chapel and C++, atomic vars default to **SC** ordering which includes both acquire and release

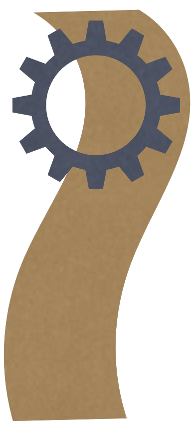
Memory Order

`atomic = 1;`
`c = atomic;`
`b = atomic;`
`atomic = 2;`

Program Order



```
atomic = 1;
c = atomic;
```



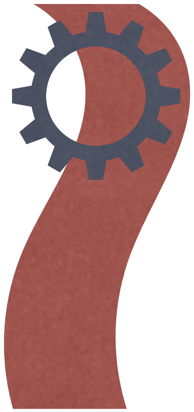
```
b = atomic;
atomic = 2;
```

Atomic vars cannot create race conditions. There is a global order. `c == 2 b == 0` not possible e.g.

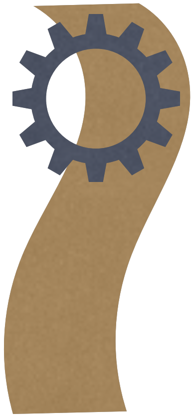
Memory Order

`atomic = 1;`
`c = atomic;`
`b = atomic;`
`atomic = 2;`

Program Order



```
atomic = 1;
c = atomic;
```



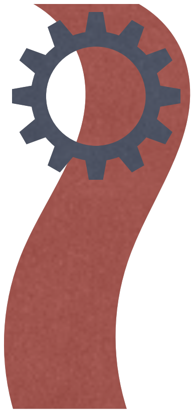
```
b = atomic;
atomic = 2;
```

Memory Order

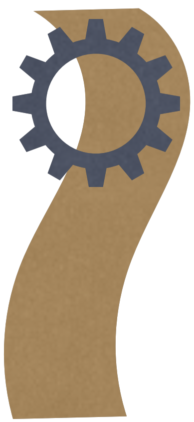
`b = atomic;`
`atomic = 1;`
`c = atomic;`
`atomic = 2;`

Program Order

Memory Order



```
x = 1;
atomic = 1;
```

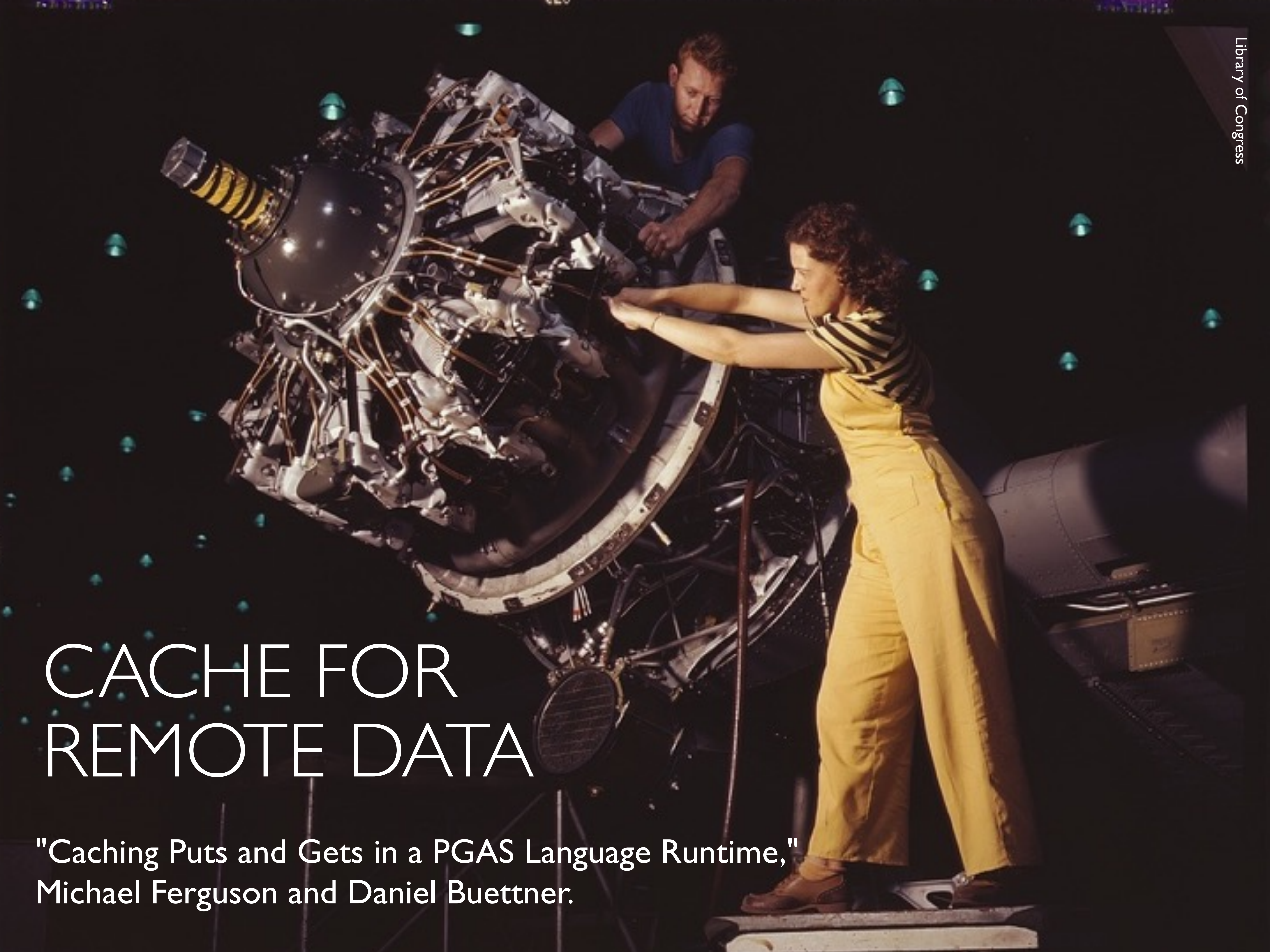


```
b = atomic;
c = x;
```

`x = 1;`
`atomic ok= 1;`
`b =atomic ok;`
`c = x;`

Atomic ops constrain the code around them

`b == 1` implies `c == 0`



Library of Congress

CACHE FOR REMOTE DATA

"Caching Puts and Gets in a PGAS Language Runtime," Michael Ferguson and Daniel Buettner.

CACHE FOR REMOTE DATA

- **Goal: communication aggregation and overlap**
- **Bonus points: avoiding repeated communication**
- **Software cache in Chapel's runtime**
- **One cache per pthread**
- **Write-back cache with dirty bits**

CACHE COHERENCY

- **Simple, local coherency**
- **Discard all cached data on *acquire***
- **Wait for pending operations on a *release***

- **Strategy used in related work with UPC**

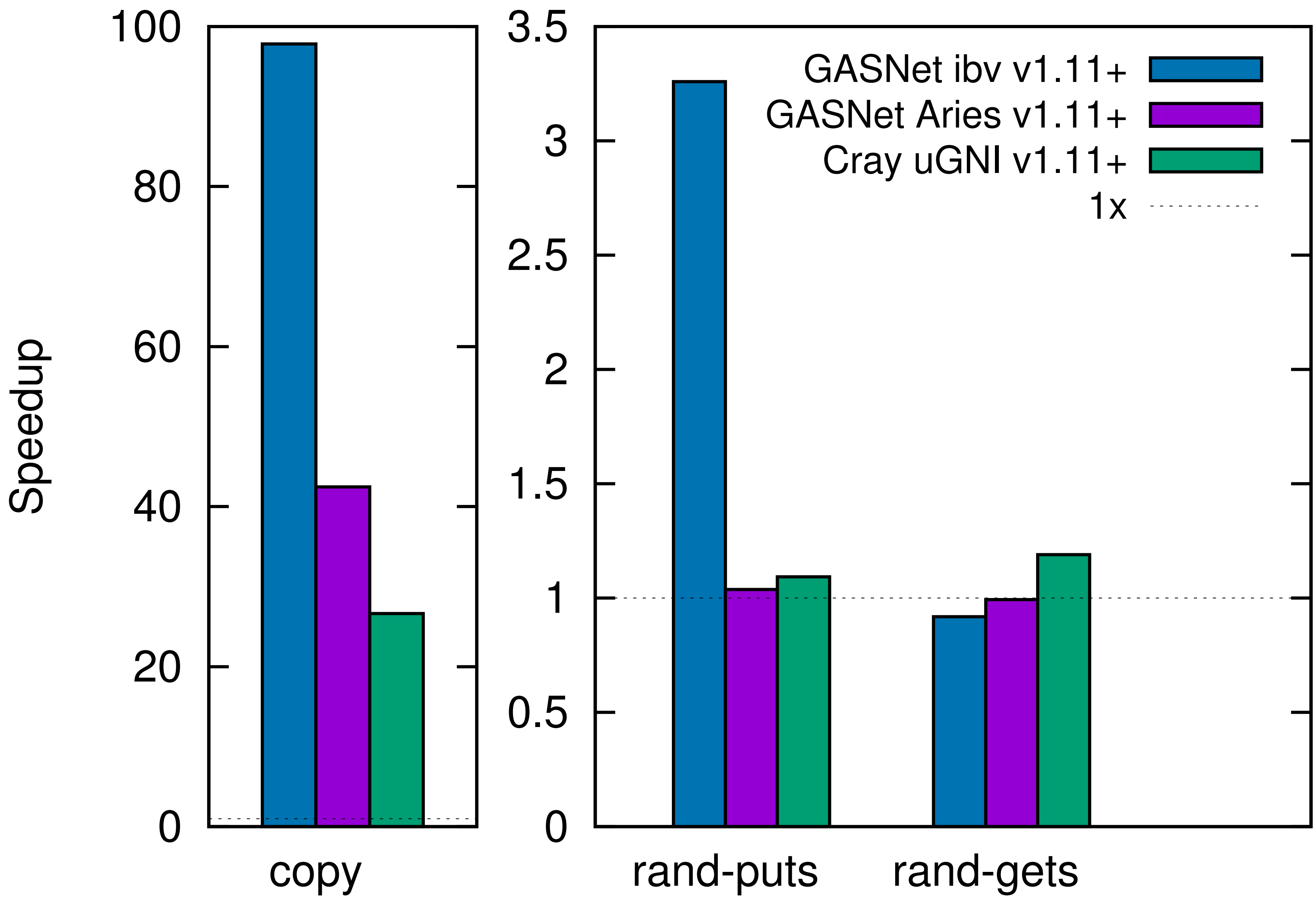
CACHE FEATURES

	Overlap		Aggregation	
	GET	PUT	GET	PUT
Do PUTs in background		X		
Start one PUT per contiguous written region				X
Round GETs up to 64-byte cache lines			X	
Sequential read-ahead	X		X	
Programmer-provided prefetch hints*	X			

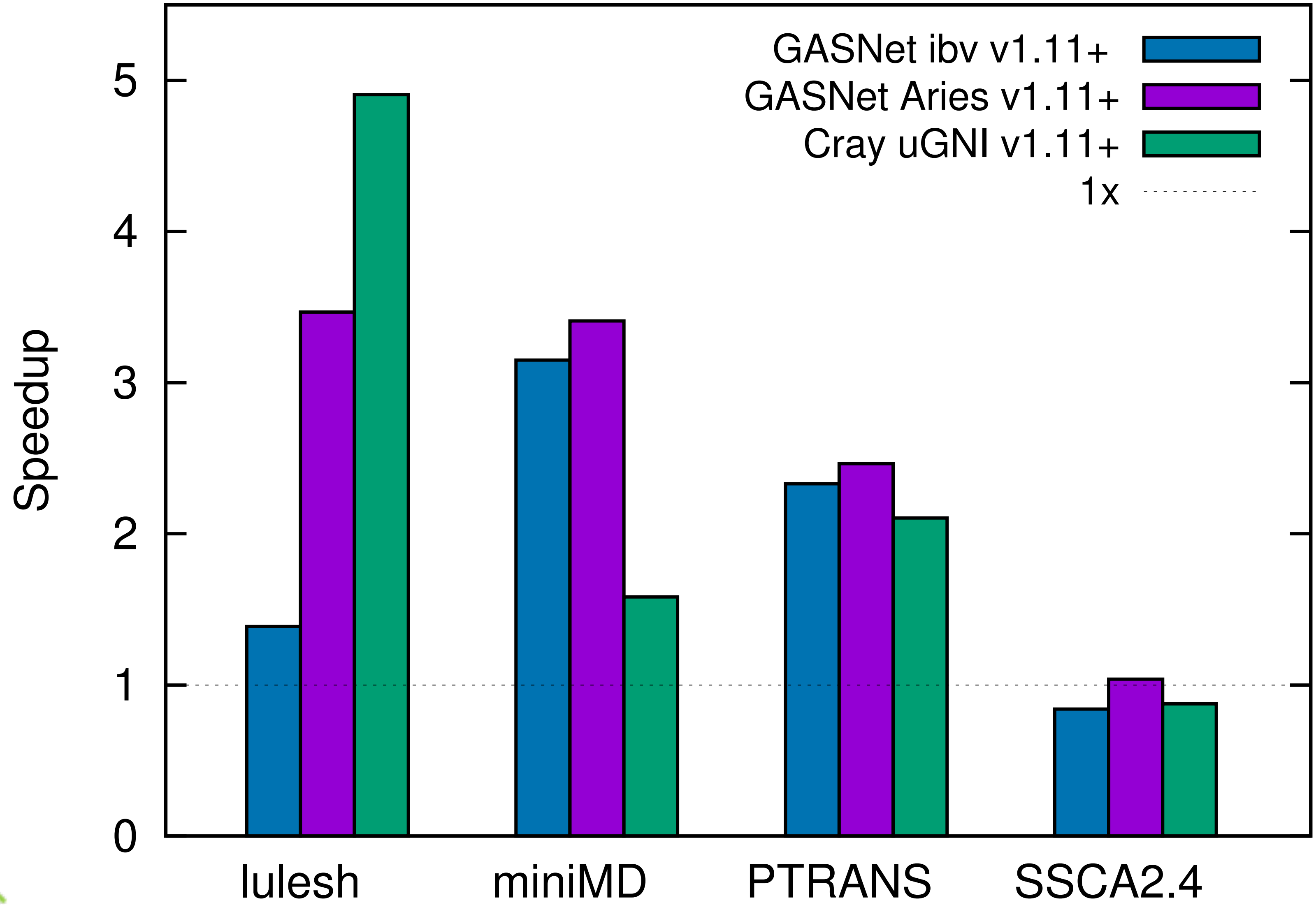
PERFORMANCE

San Diego Air and Space Museum

SYNTHETIC BENCHMARKS



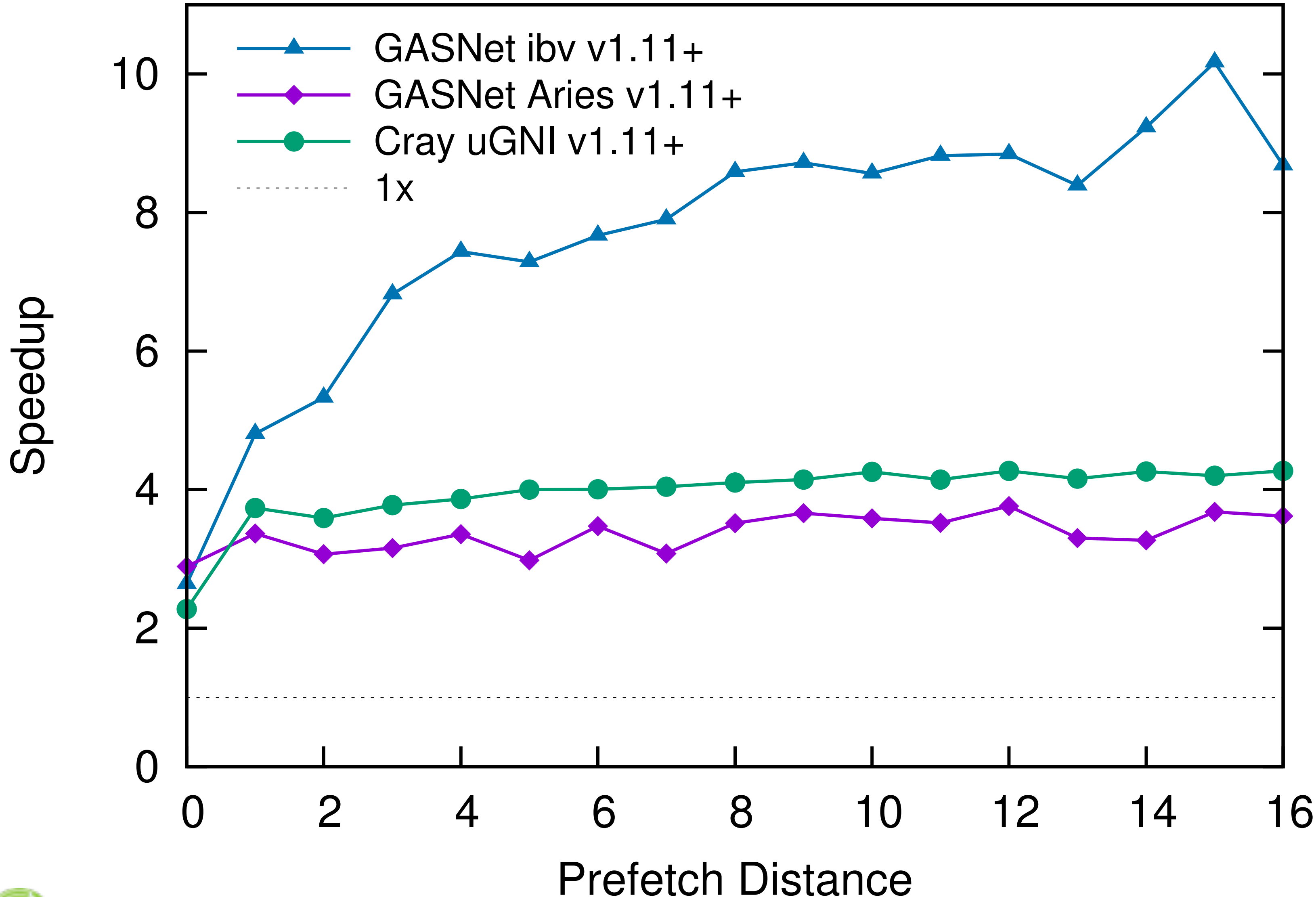
APPLICATION BENCHMARKS



PREFETCH EXAMPLE

```
var A:[1..n] int;
on Locales[1] {
   var sum:int;
   // Optional warm up
   for i in 1..k do prefetch(A[f(i)]);
   for i in 1..n {
   if i+k <= n then prefetch(A[f(i+k)]);
       sum += A[f(i)]
   }
}
```

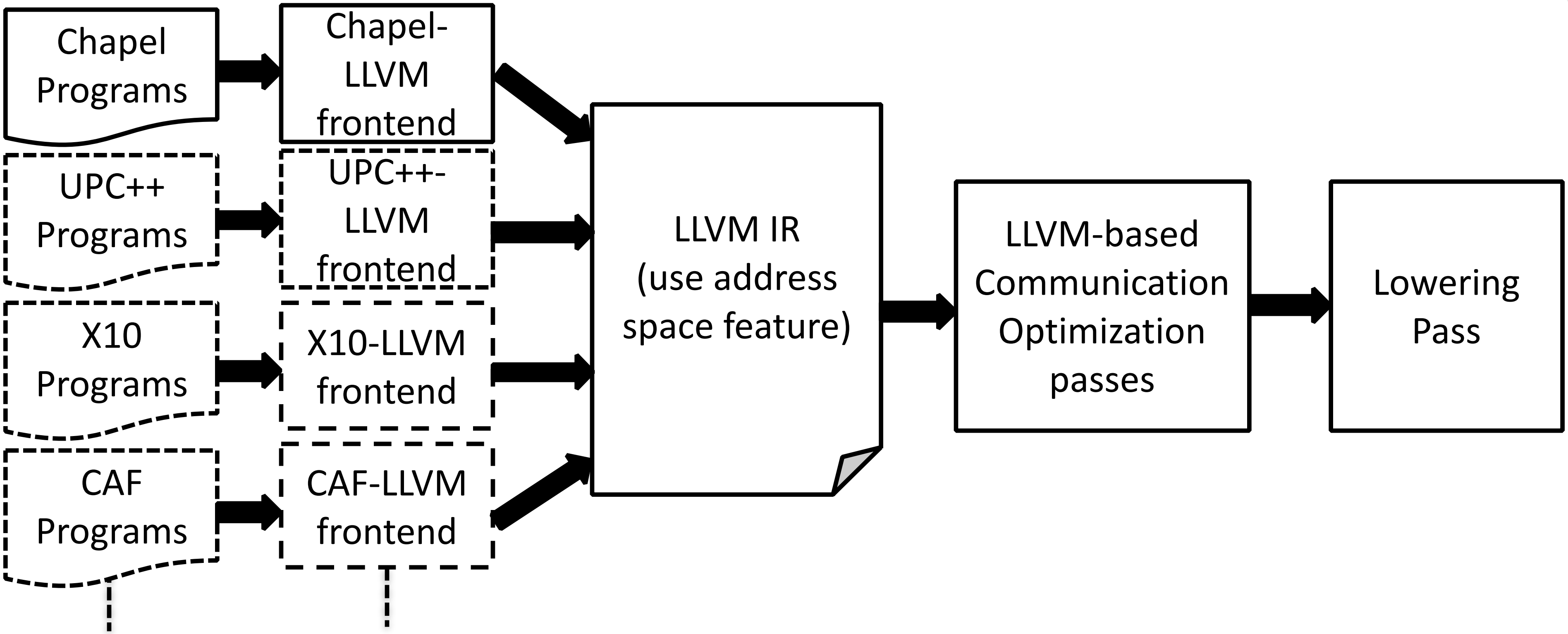
PREFETCH EXAMPLE



OPTIMIZING COMMUNICATION WITH LLVM

"LLVM-based Communication Optimizations for PGAS Programs"
Akihito Hayashi, Jisheng Zhao, Michael Ferguson, Vivek Sarkar

THE VISION:
SHARED PGAS OPTIMIZATION PASSES



EXAMPLE

```
// x is remote
var sum = 0;
for i in 1..100 {
  sum += get(x);
}
```

```
// x is possibly remote
var sum = 0;
for i in 1..100 {
  %1 = get(x);
  sum += %1;
}
```

TO GLOBAL MEMORY

```
var sum = 0;
for i in 1..100 {
  %1 = load <100> %x
  sum += %1;
}
```

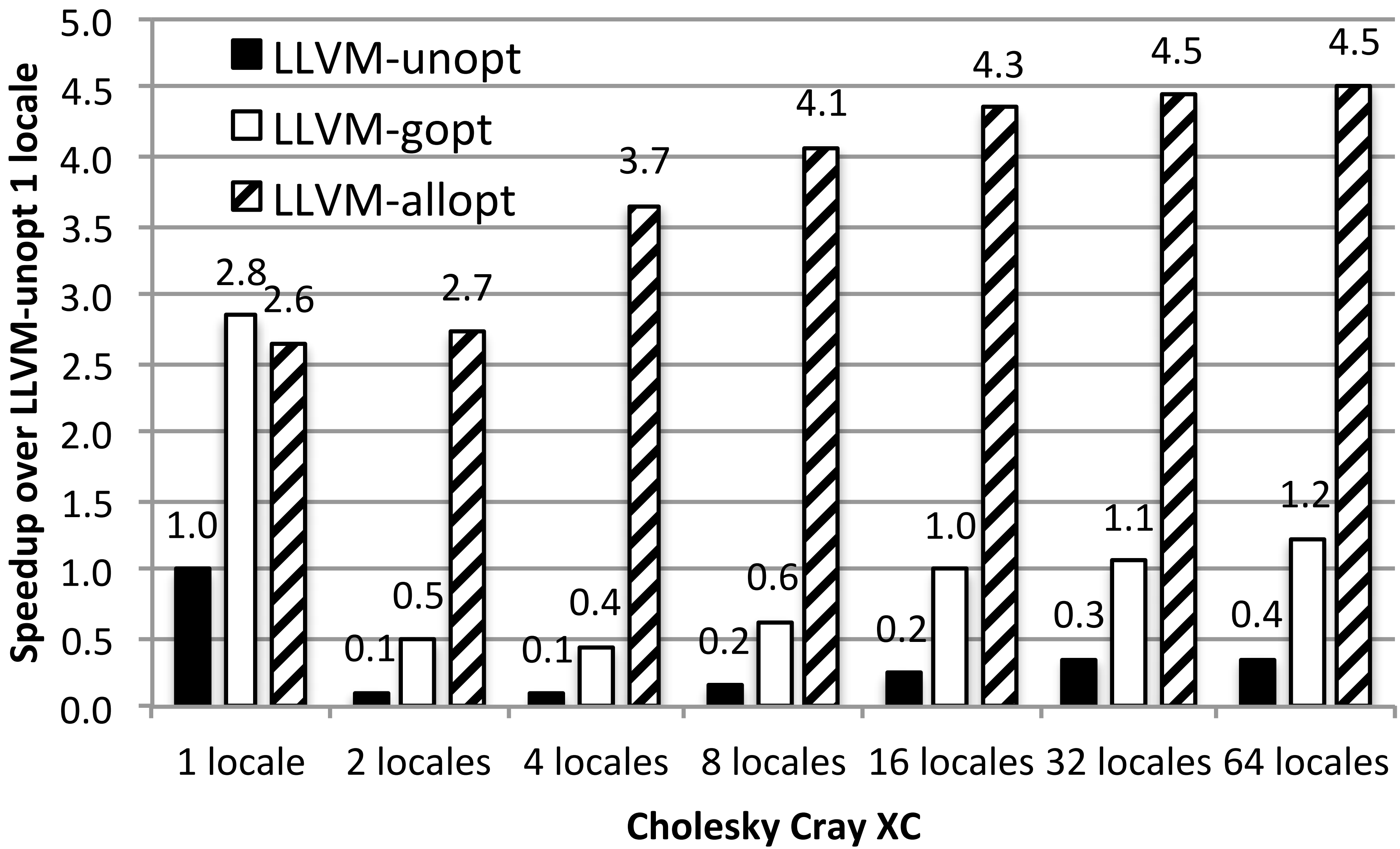
EXISTING LLVM OPTIMIZATION LICM

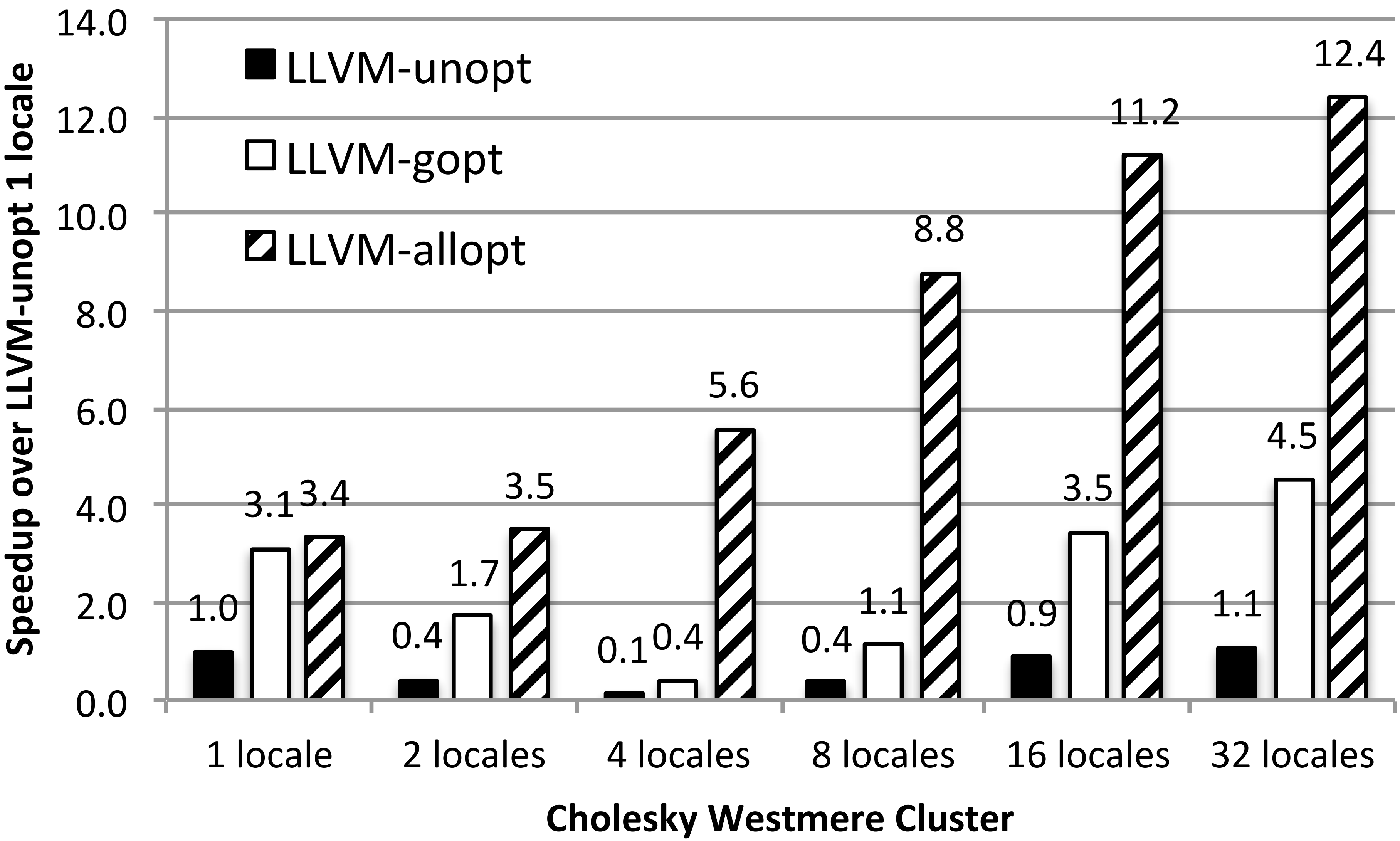
```
// existing LLVM opt
var sum = 0;
%1 = load <100> %x
for i in 1..100 {
  sum += %r1;
}
```

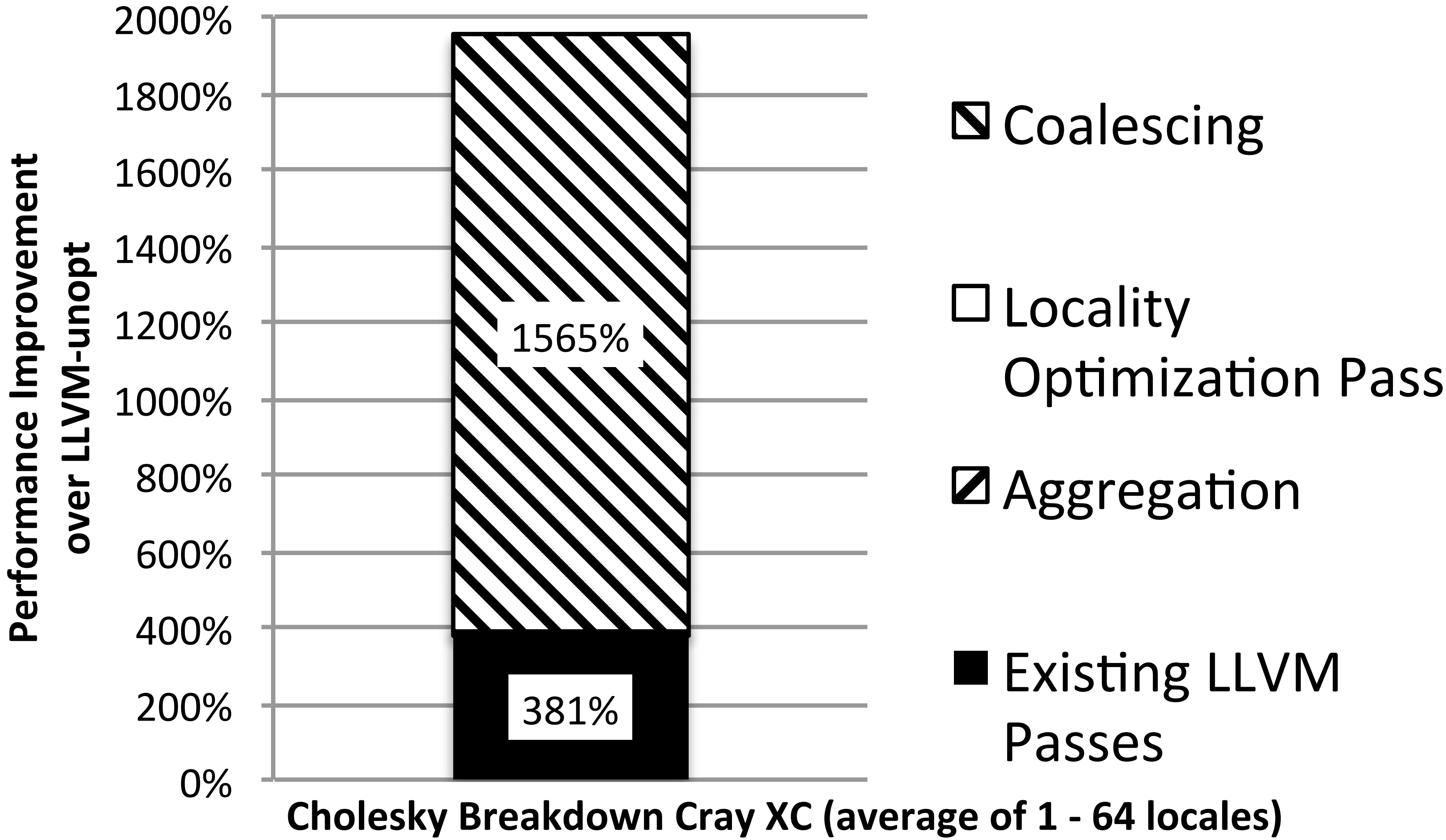
TO DISTRIBUTED MEMORY

```
var sum = 0;
%1 = get(x);
for i in 1..100 {
  sum += %1;
}
```

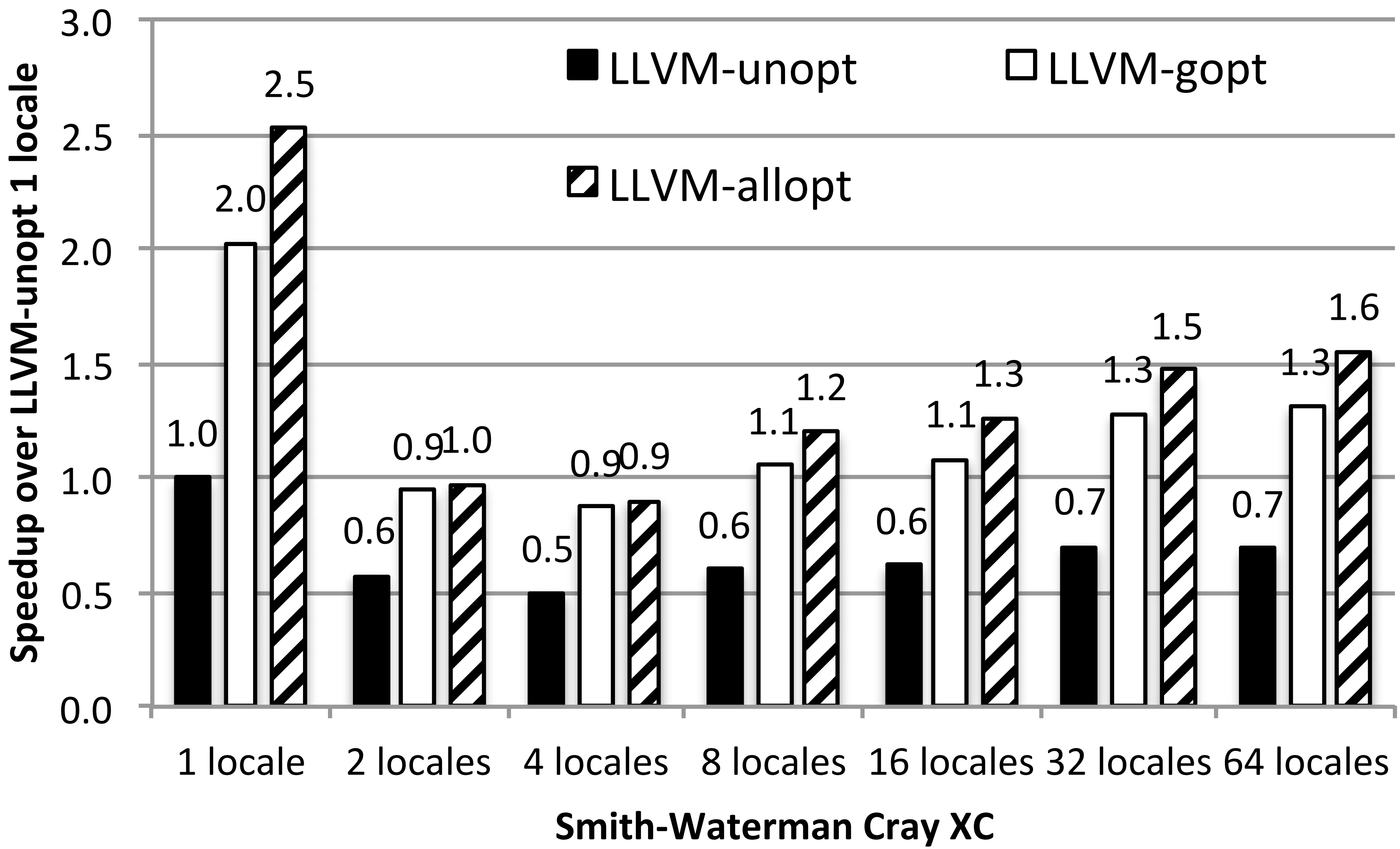
```
load <100> %x = load i64 addrspace(100)* %x
```


5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Speedup over LLVM-unopt 1 locale
LLVM-unopt
LLVM-gopt
LLVM-allopt
1.0
2.8
2.6
0.1
0.5
2.7
0.1
0.4
3.7
0.2
0.6
4.1
0.2
1.0
4.3
0.3
1.1
4.5
0.4
1.2
4.5
1 locale
2 locales
4 locales
8 locales
16 locales
32 locales
64 locales
Cholesky Cray XC

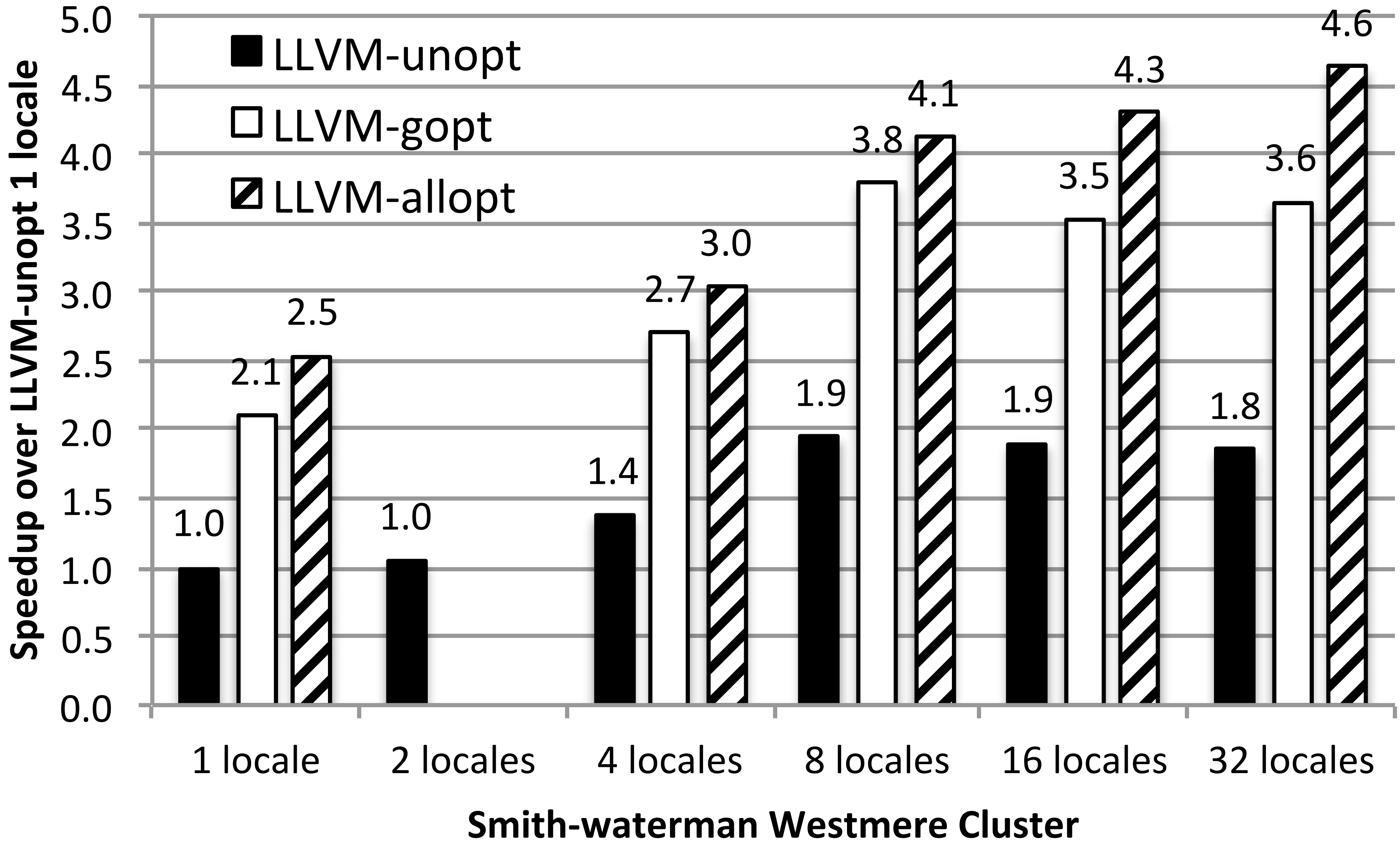

LLVM-unopt
LLVM-gopt
LLVM-allopt
Speedup over LLVM-unopt 1 locale
14.0
12.0
10.0
8.0
6.0
4.0
2.0
0.0
1.0
3.1
3.4
0.4
1.7
3.5
0.1
0.4
5.6
0.4
1.1
8.8
0.9
3.5
11.2
1.1
4.5
12.4
1 locale
2 locales
4 locales
8 locales
16 locales
32 locales
Cholesky Westmere Cluster

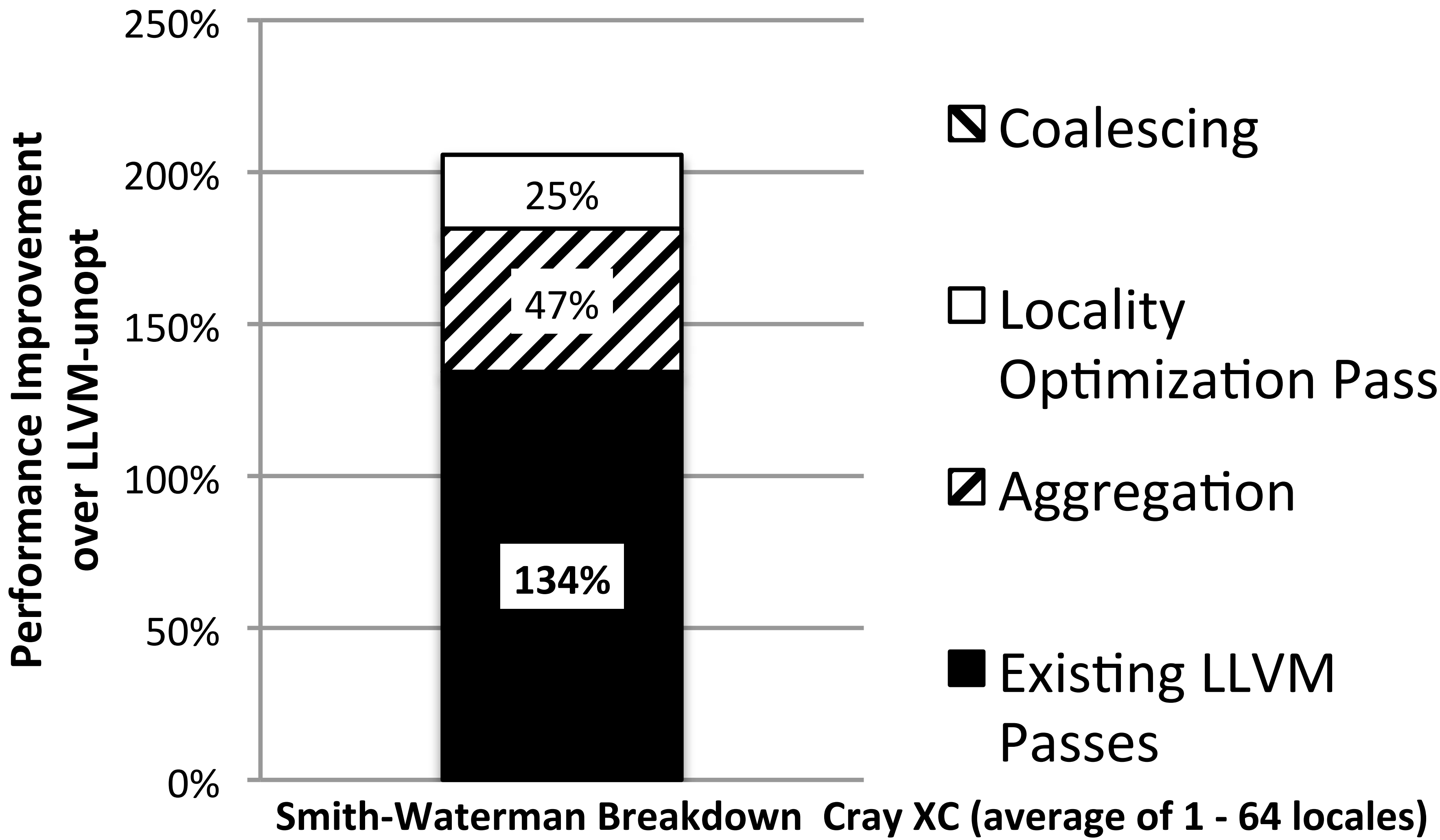


Performance Improvement over LLVM-unopt
2000%
1800%
1600%
1400%
1200%
1000%
800%
600%
400%
200%
0%
1565%
381%
Coalescing
Locality Optimization Pass
Aggregation
Existing LLVM Passes
Cholesky Breakdown Cray XC (average of 1 - 64 locales)

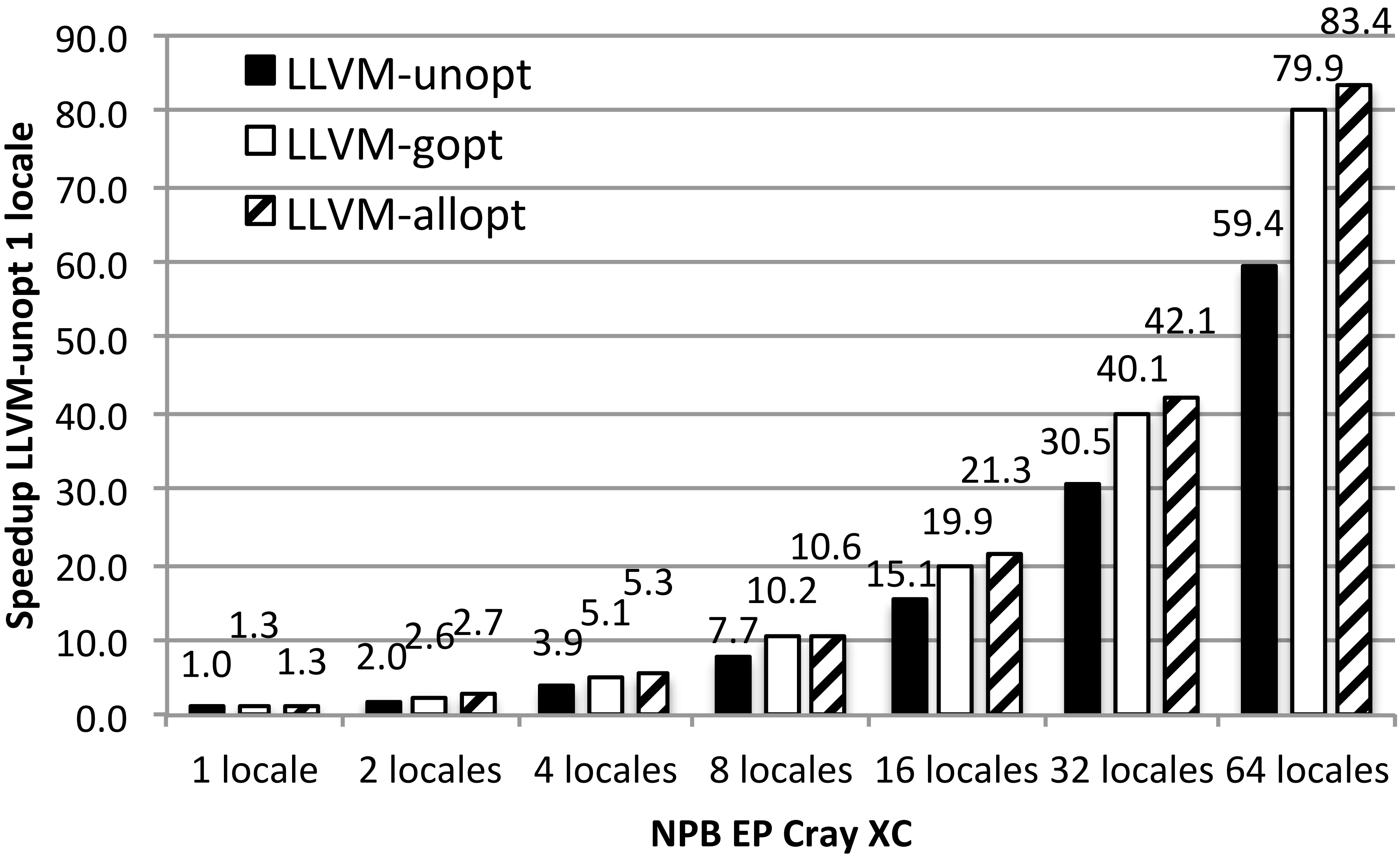


LLVM-unopt
LLVM-gopt
LLVM-allopt
Speedup over LLVM-unopt 1 locale
3.0
2.5
2.0
1.5
1.0
0.5
0.0
1.0
2.0
2.5
0.6
0.9
1.0
0.5
0.9
0.9
0.6
1.1
1.2
0.6
1.1
1.3
0.7
1.3
1.5
0.7
1.3
1.6
1 locale
2 locales
4 locales
8 locales
16 locales
32 locales
64 locales
Smith-Waterman Cray XC

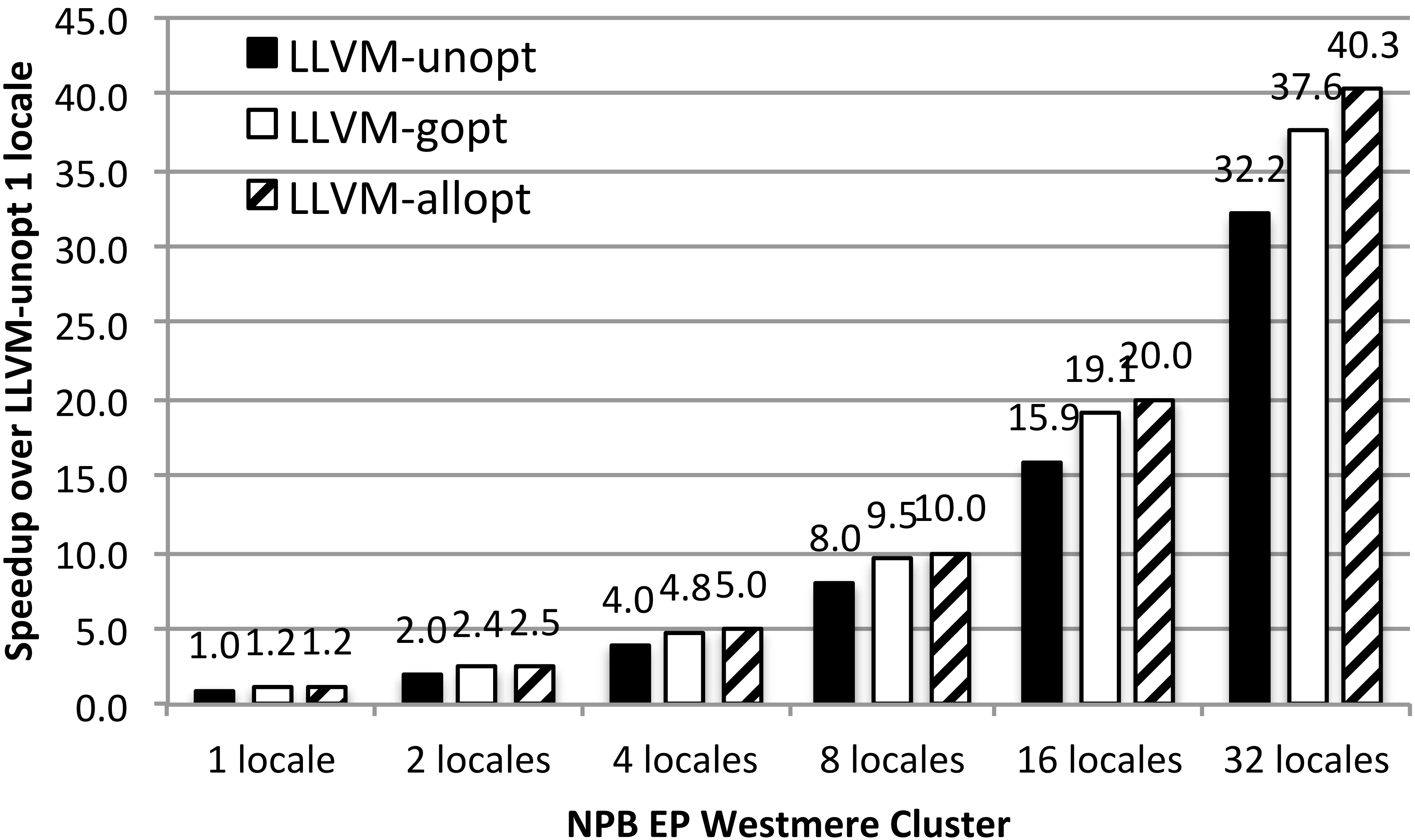

Speedup over LLVM-unopt 1 locale
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
LLVM-unopt
LLVM-gopt
LLVM-allopt
1.0
2.1
2.5
1.0
1.4
2.7
3.0
1.9
3.8
4.1
1.9
3.5
4.3
1.8
3.6
4.6
1 locale
2 locales
4 locales
8 locales
16 locales
32 locales
Smith-waterman Westmere Cluster



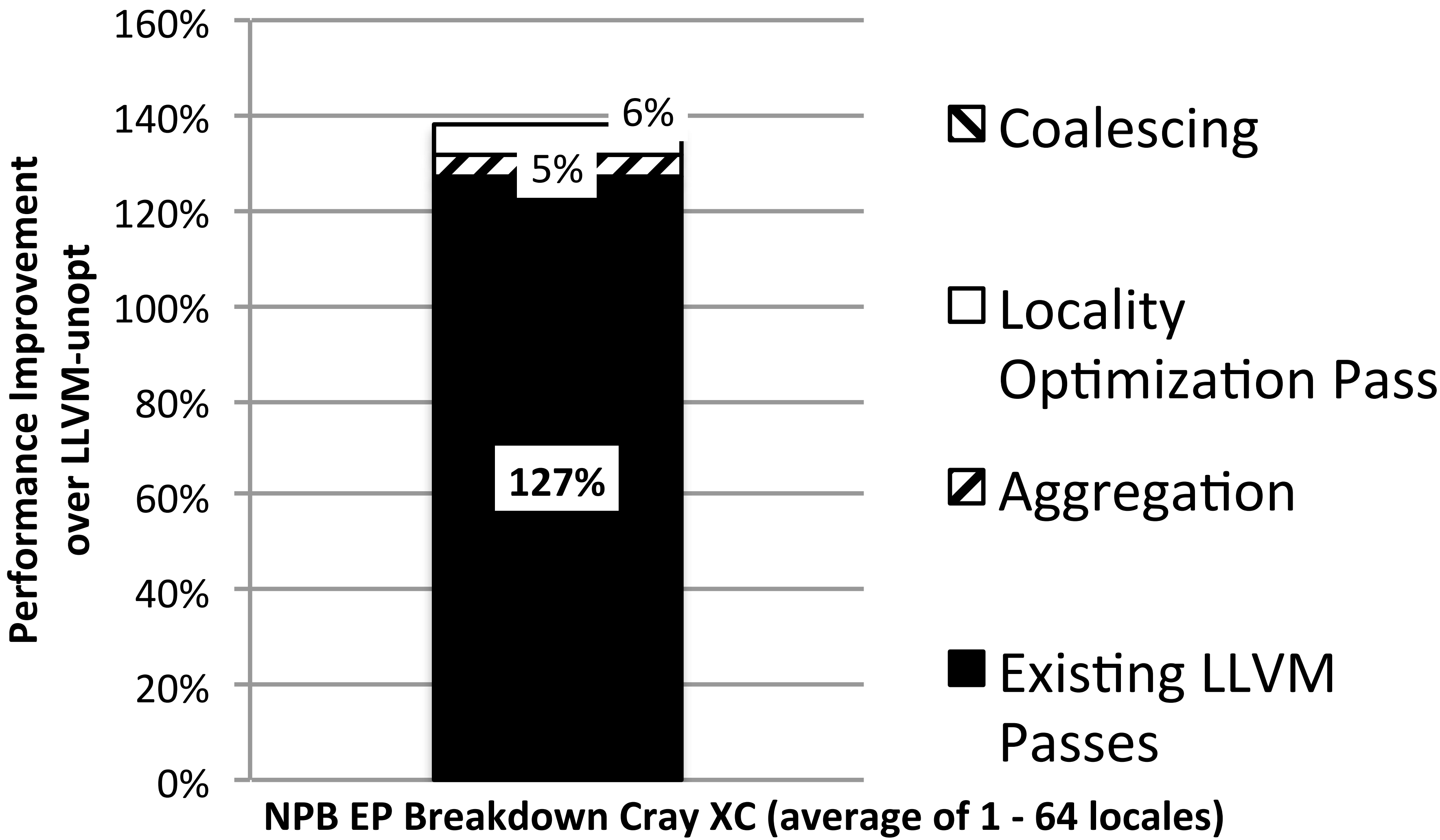
250%
200%
150%
100%
50%
0%
Performance Improvement over LLVM-unopt
25%
47%
134%
Coalescing
Locality Optimization Pass
Aggregation
Existing LLVM Passes
Smith-Waterman Breakdown Cray XC (average of 1 - 64 locales)



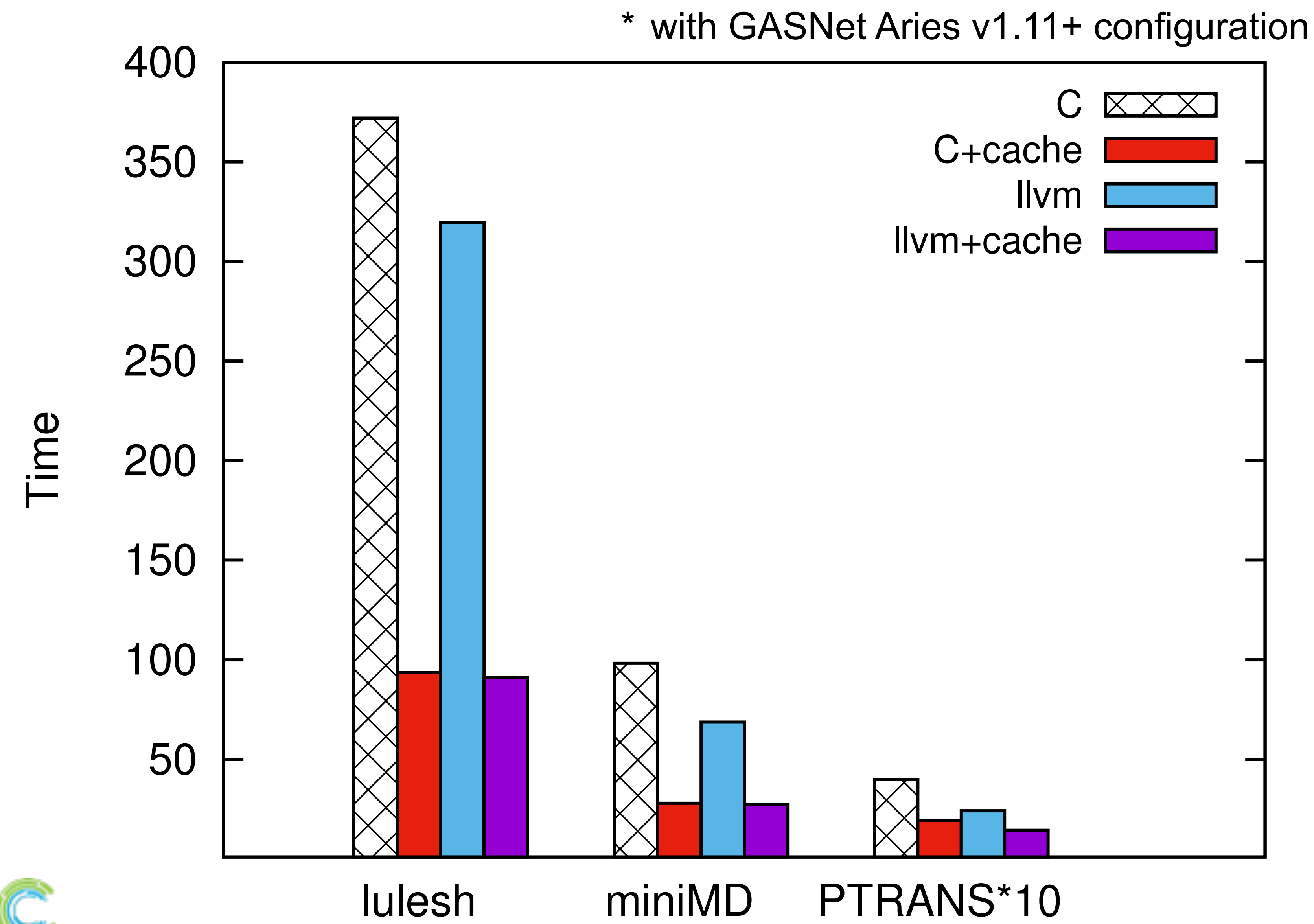
Speedup LLVM-unopt 1 locale
LLVM-unopt
LLVM-gopt
LLVM-allopt
90.0
80.0
70.0
60.0
50.0
40.0
30.0
20.0
10.0
0.0
1.0
1.3
1.3
2.0
2.6
2.7
3.9
5.1
5.3
7.7
10.2
10.6
15.1
19.9
21.3
30.5
40.1
42.1
59.4
79.9
83.4
1 locale
2 locales
4 locales
8 locales
16 locales
32 locales
64 locales
NPB EP Cray XC



Speedup over LLVM-unopt 1 locale
45.0
40.0
35.0
30.0
25.0
20.0
15.0
10.0
5.0
0.0
LLVM-unopt
LLVM-gopt
LLVM-allopt
1.0
1.2
1.2
2.0
2.4
2.5
4.0
4.8
5.0
8.0
9.5
10.0
15.9
19.1
20.0
32.2
37.6
40.3
1 locale
2 locales
4 locales
8 locales
16 locales
32 locales
NPB EP Westmere Cluster


Performance Improvement over LLVM-unopt
160%
140%
120%
100%
80%
60%
40%
20%
0%
6%
5%
127%
Coalescing
Locality Optimization Pass
Aggregation
Existing LLVM Passes
NPB EP Breakdown Cray XC (average of 1 - 64 locales)

CACHING VS LLVM



Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, URIKA, and YARCDATA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.

http://chapel.cray.com chapel_info@cray.com https://github.com/chapel-lang/chapel/