Enabling Parallel Computing in Chapel with Clang and LLVM

Michael Ferguson
Cray Inc.

October 19, 2017

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.

Outline

- Introducing Chapel
- C interoperability
- Combined Code Generation
- Communication Optimization in LLVM

What is Chapel?

Chapel: A productive parallel programming language

- portable
- open-source
- a collaborative effort

Goals:

- Support general parallel programming
  - “any parallel algorithm on any parallel hardware”
- Make parallel programming at scale far more productive

What does “Productivity” mean to you?

Recent Graduates:

“something similar to what I used in school: Python, Matlab, Java, …”

Seasoned HPC Programmers:

“that sugary stuff that I don’t need because I ~~was born to suffer”~~

want full control to ensure performance”

Computational Scientists:

“something that lets me express my parallel computations without having to wrestle with architecture-specific details”

Chapel Team:

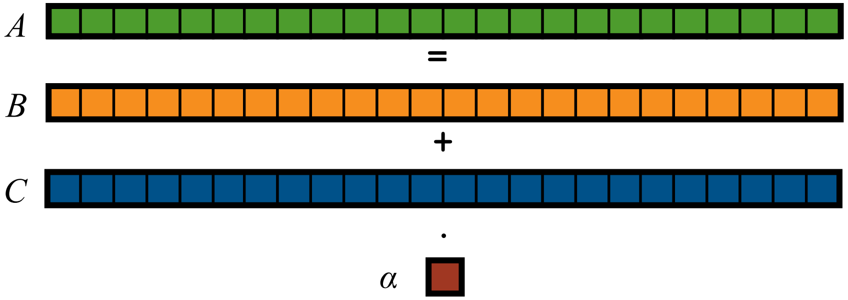
“something that lets computational scientists express what they want, without taking away the control that HPC programmers want, implemented in a language as attractive as recent graduates want.”

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:

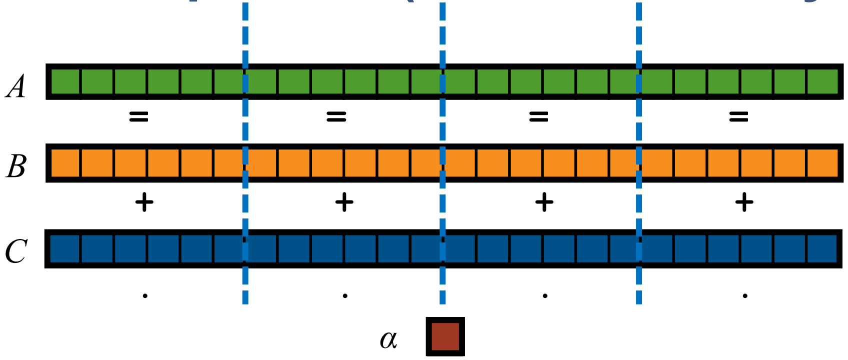


STREAM Triad: a trivial parallel computation

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 (shared memory / multicore):

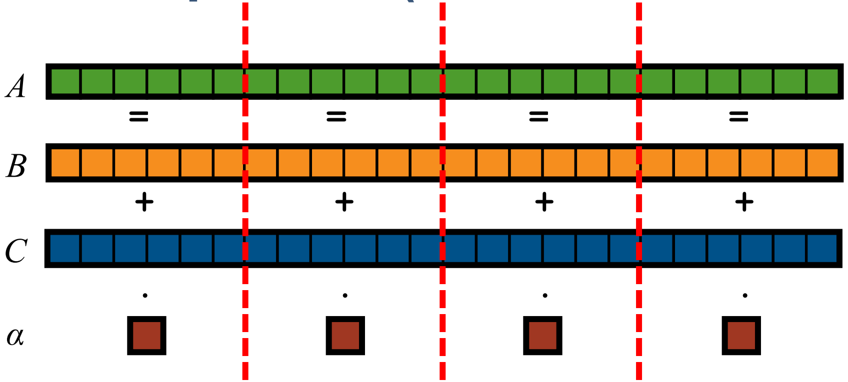


STREAM Triad: a trivial parallel computation

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):

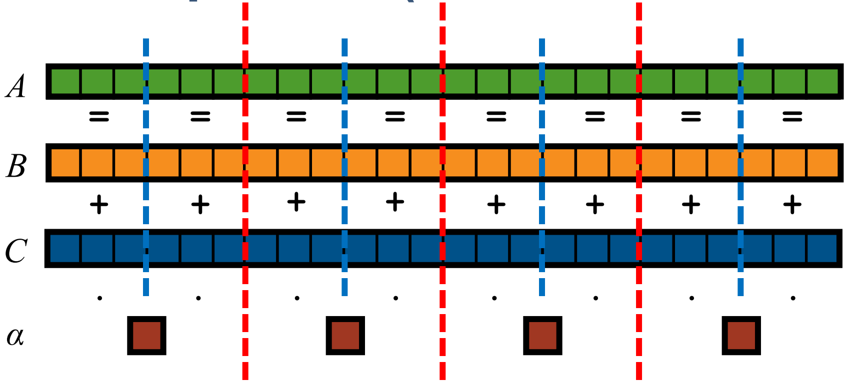


STREAM Triad: a trivial parallel computation

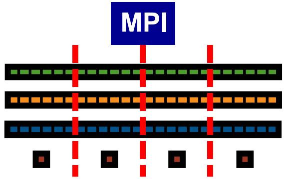
Given: m-element vectors A, B, C

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

In pictures, in parallel (distributed memory multicore):



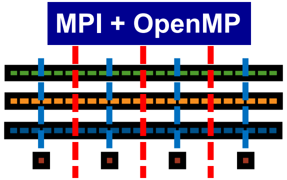
STREAM Triad: MPI




```
#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);

return 0;  }
```

STREAM Triad: MPI+OpenMP

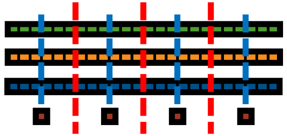



```c
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

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

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

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

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

  return errCount;
}

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

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

  a = HPCC_XMALLOC( double, VectorSize );
  b = HPCC_XMALLOC( double, VectorSize );
  c = HPCC_XMALLOC( double, VectorSize );
```

```c
  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;  }
```

STREAM Triad: MPI+OpenMP

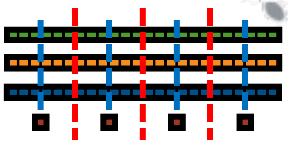
MPI + OpenMP



```
#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);

  return 0;  }
```

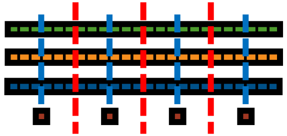
CUDA



```
#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

  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];  }
```

STREAM Triad: MPI+OpenMP

MPI + OpenMP



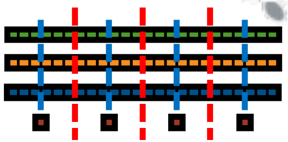
```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

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;
```

```
  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;
  }
```

```
  }
  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 );
```

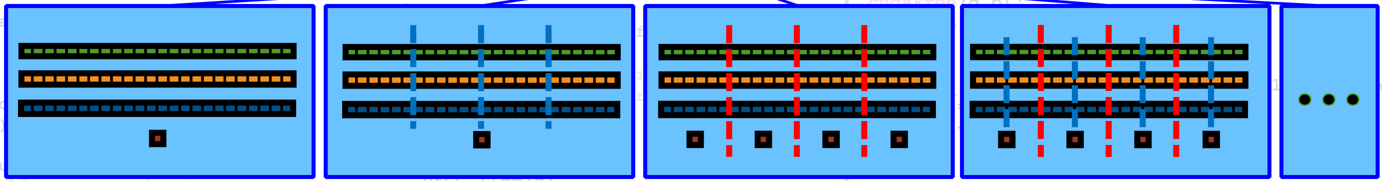
```
  cudaThreadSynchronize();

  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.
This tends to be a result of bottom-up language design.

STREAM Triad: Chapel

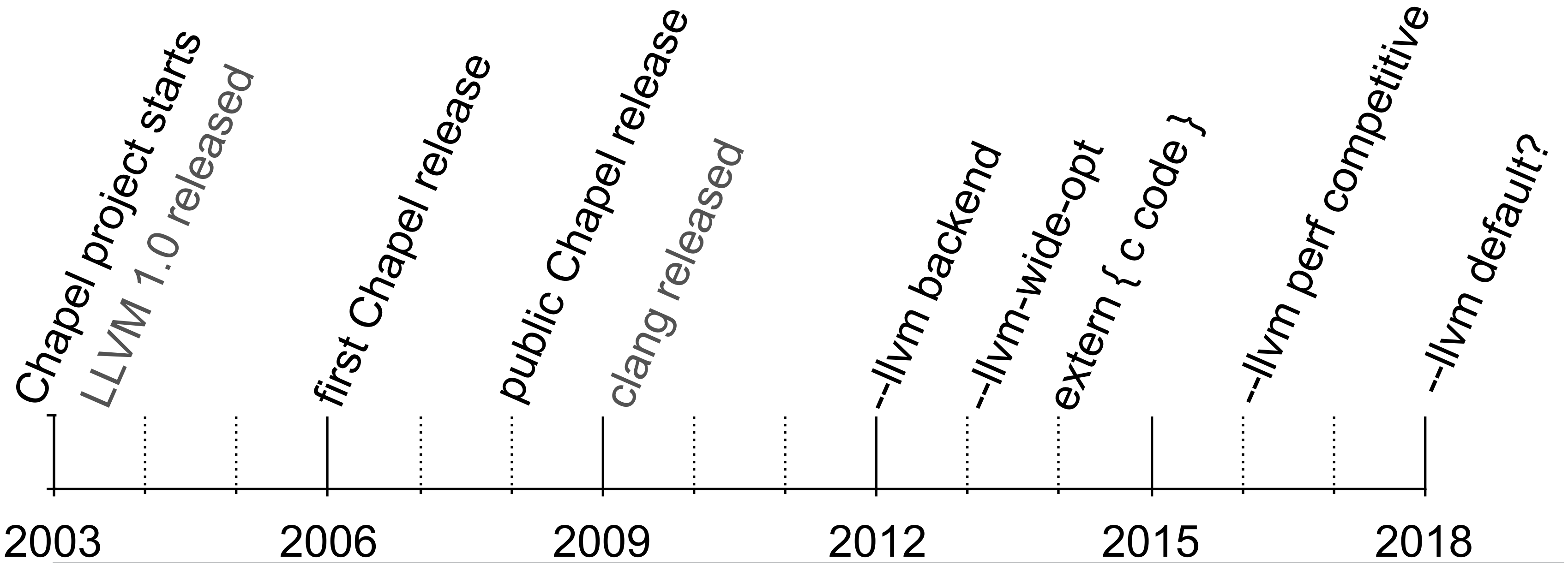
```
use …;

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: How should this index set—and any arrays and computations over it—be mapped to the system?



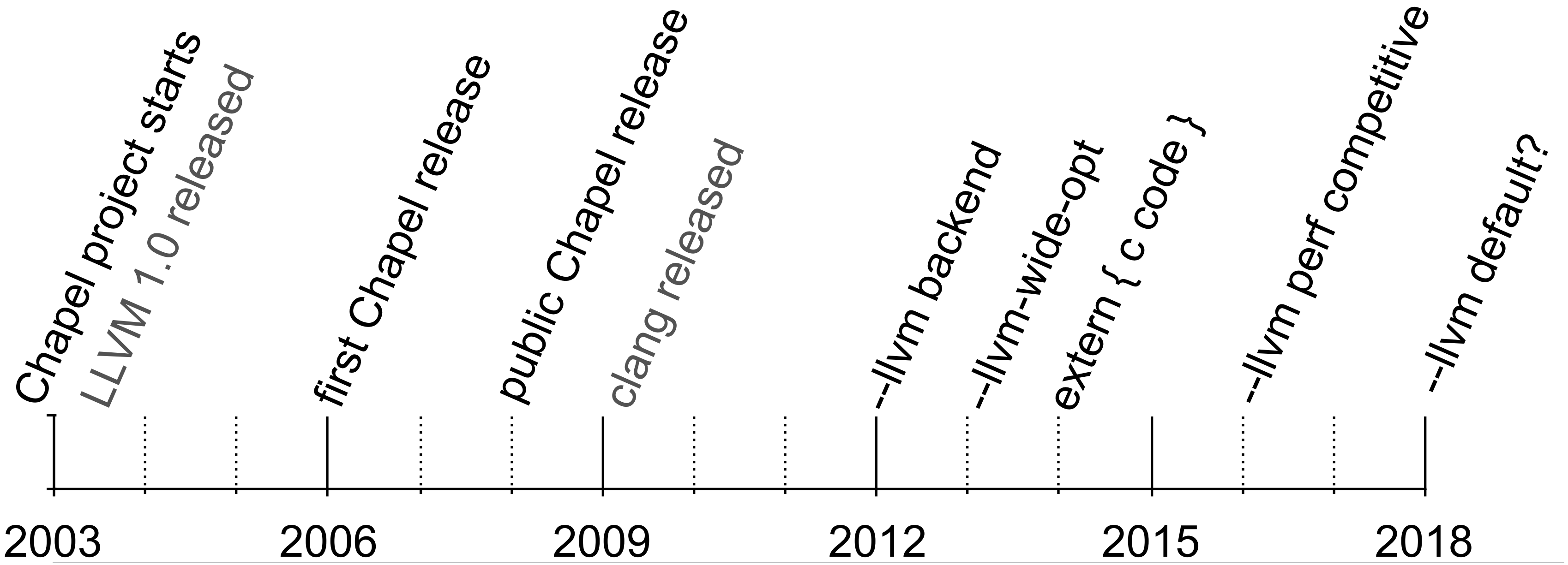
Philosophy: Good, *top-down* language design can tease system-specific implementation details away from an algorithm, permitting the compiler, runtime, applied scientist, and HPC expert to each focus on their strengths.

Chapel+LLVM History

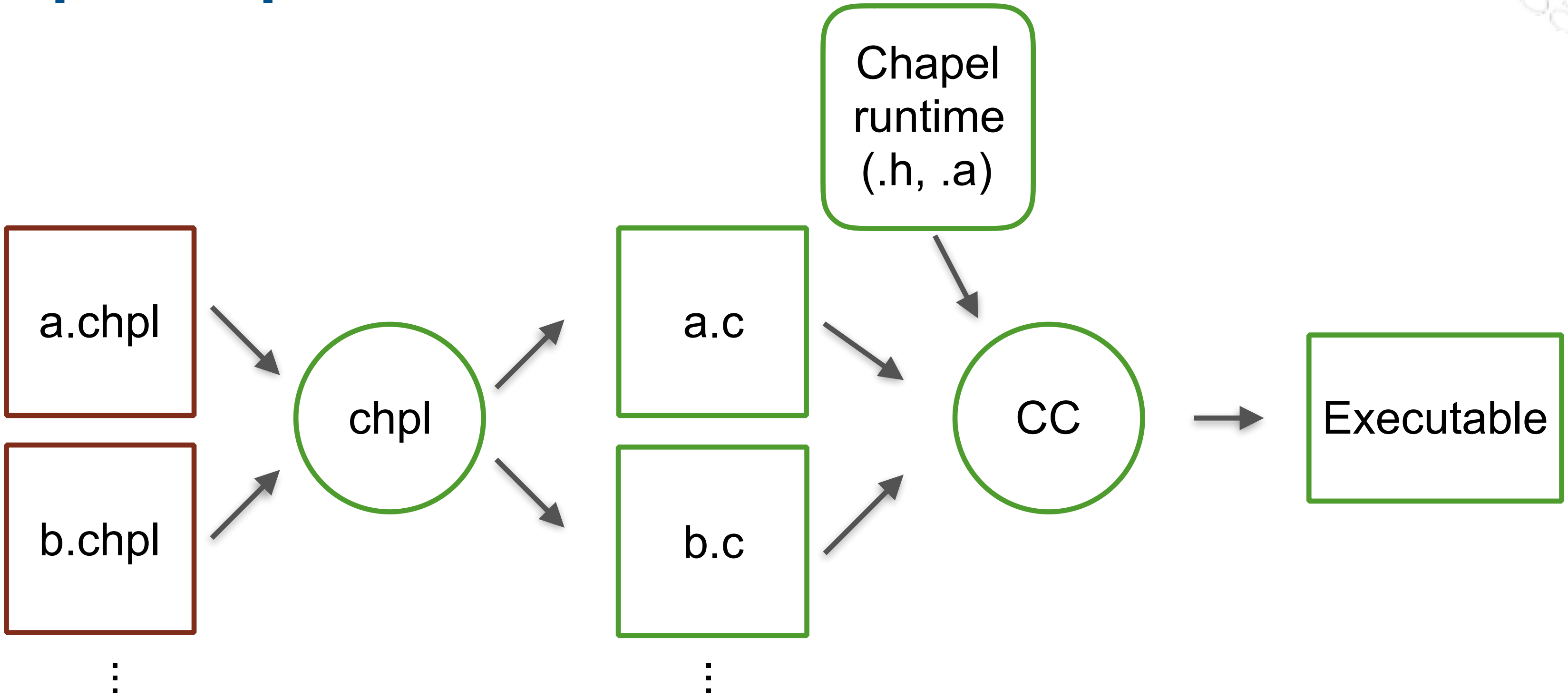


Chapel+LLVM History

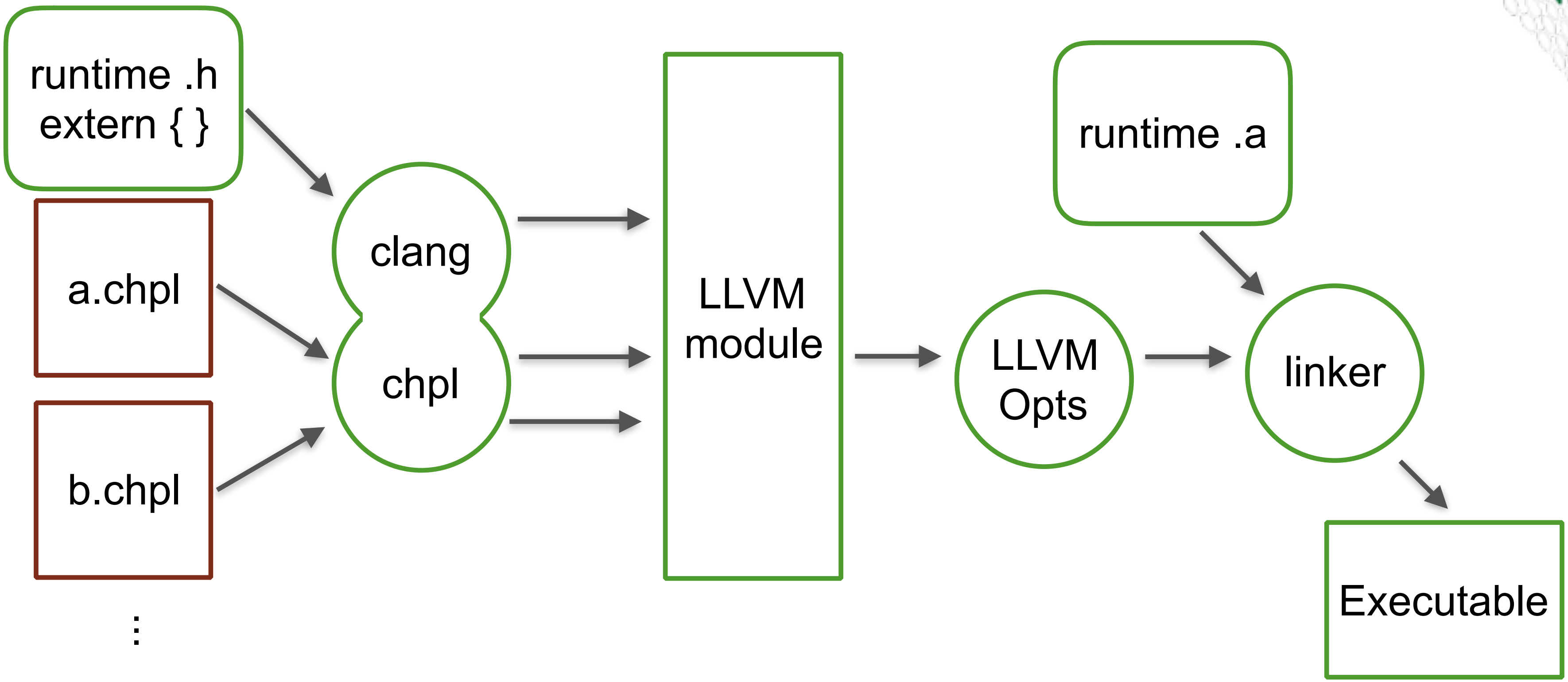
- Chapel project grew up generating C code



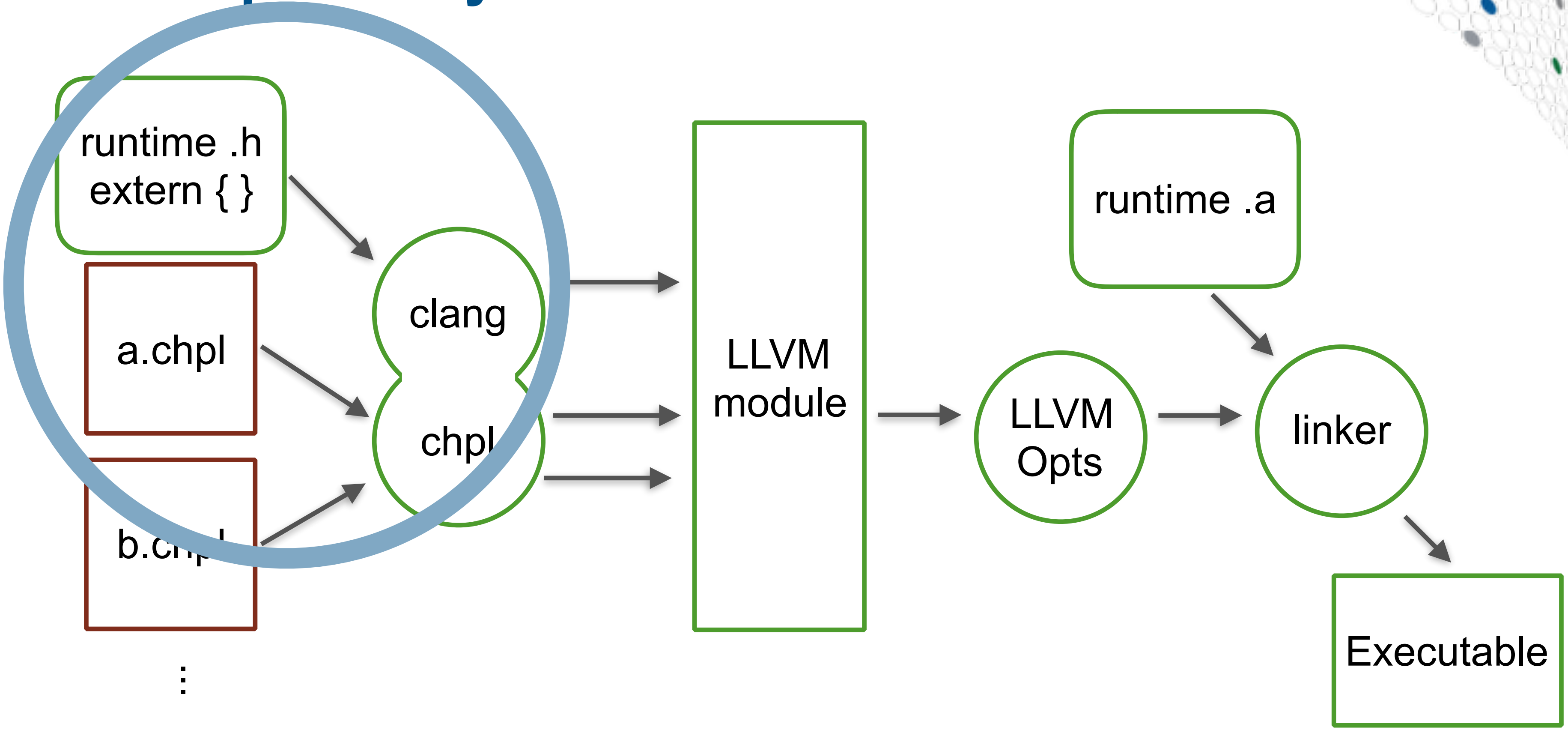
chpl compilation flow



chpl --llvm compilation flow



C Interoperability



Goals of C Interoperability

- Chapel is a new language
- Libraries are important for productivity

➡ Easy use of libraries in another language is important!

- Chapel supports interoperability with C
- Need to be able to use an existing C library
  - functions, variables, types, and macros
- Using C functions needs to be efficient
  - performance is a goal here!

C Interoperability Example

// add1.h

```
static inline
int add1(int x) { return x+1; }
```

// addone.chpl

```
extern proc add1(x:c_int):c_int;
writeln(add1(4));
```

```
$ chpl addone.chpl add1.h
$ ./addone
5
```

With extern { }

// add1.h

```
static inline
int add1(int x) { return x+1; }
```

// addone.chpl

```
extern { #include “add1.h” }
writeln(add1(4));
```

```
$ chpl addone.chpl
$ ./addone
5
```

extern block compilation flow

frontend passes

parse .chpl

readExternC

⋮

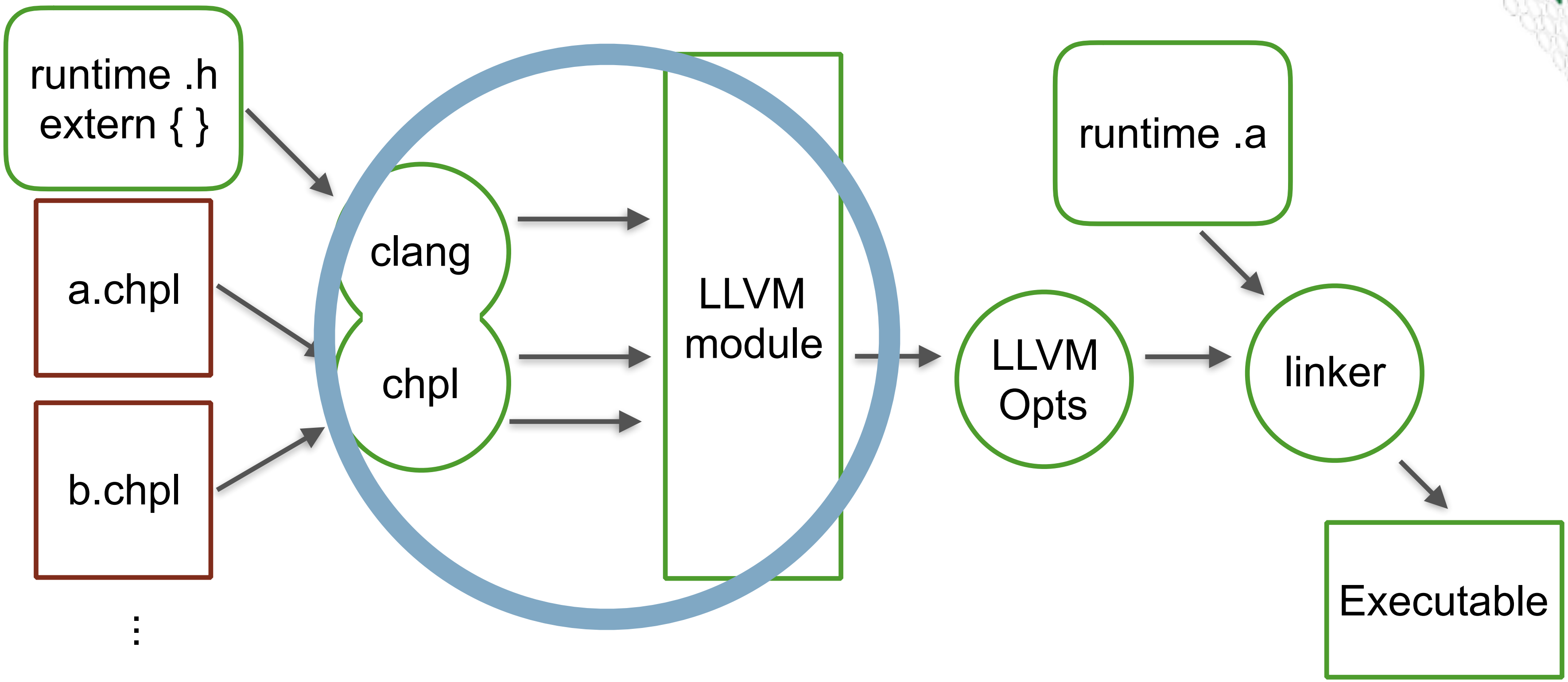
extern {
  int add1(**int** x);
}

clang parse:
C ➡ clang AST

readExternC:
clang AST ➡ chapel extern decls

extern proc add1(x:**c_int**):**c_int**;

Combined Code Generation



Inlining with C code

- Some C functions expect to be inlined
  - if not, there is a performance penalty
- Runtime is written primarily in C
  - Enabling easy use of libraries like qthreads
- Chapel uses third-party libraries such as GMP
  - Library authors control what might be inlined
- Since Chapel generated C, it became normal to assume:
  - functions can be inlined
  - C types are available
  - fields in C structs are available
  - C macros are available

Example: Accessing a Struct Field

// sockaddr.h

```
#include <sys/socket.h>
typedef struct my_sockaddr_s {
  struct sockaddr_storage addr;
  size_t len;
} my_sockaddr_t;
```

// network.chpl

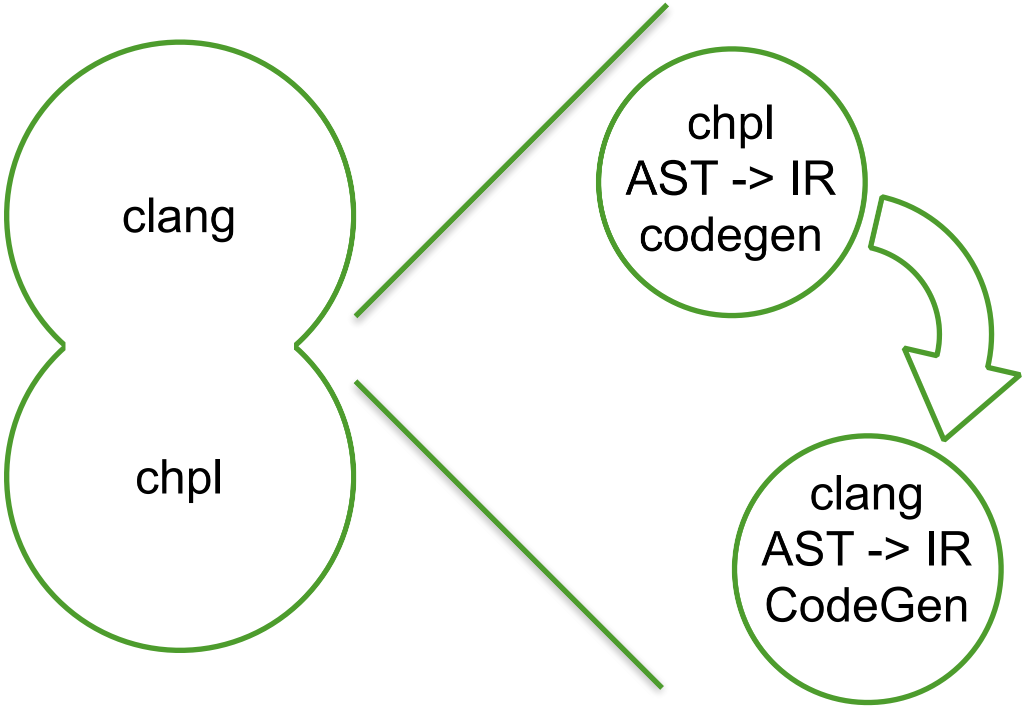
```
require "sockaddr.h";
extern record my_sockaddr_t {
  var len: size_t;
}
```

OR

```
extern { #include “sockaddr.h” }
```

```
var x:my_sockaddr_t;
x.len = c_sizeof(c_int);
writeln(x.len);
```

Implementing Combined Code Generation



Call to C proc? Use of C global?
  GetAddrOfGlobal
Use of extern type?
  *CodeGen::convertTypeForMemory**
Use of field in extern record?
  *CodeGen::getLLVMFieldNumber***

* *new to clang 5*
** *will be new in clang 6*

C Macros

// usecs.h

```
#define USECS_PER_SEC 1000000
```

// microseconds.chpl

```
require "usecs.h";
const USECS_PER_SEC:c_int;
config const secs = 1;
writeln(secs, " seconds is ",
        secs*USECS_PER_SEC,
        " microseconds");
```

```
$ chpl macrodemo.chpl
$ ./macrodemo
5
```

How can this work when we generate LLVM IR?

'forall' parallelism

- The earlier example used

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

- Which is equivalent to:

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

- Chapel's forall loop represents a data parallel loop
  - iterations can run in any order
  - typically divides iterations up among some tasks
  - parallelism is controlled by what is iterated over

'forall' lowering (1)

```
// user-code.chpl
forall x in MyIter() {
  body(x);
}
```

\+

```
// library.chpl
iter MyIter(...) {
  coforall i in 1..n do  // creates n tasks
    for j in 1..m do
      yield i*m+j;    }
```

```
coforall i in 1..n do
  for j in 1..m do
    body(i*m+j);
```

'forall' lowering (2)

```
coforall i in 1..n do
  for j in 1..m do
    body(i*m+j);
```

```
count = n
for i in 1..n do
  spawn(taskfn, i)
wait for count == 0
```

```
proc taskfn(i) {
  for j in 1..m do
    body(i*m+j);
}
```

Could be represented in a parallel LLVM IR

Vectorizing

- We'd like to vectorize 'forall' loops
- Recall, 'forall' means iterations can run in any order
- Two strategies for vectorization:

A. Vectorize in Chapel front-end
   - Chapel front-end creates vectorized LLVM IR
   - Challenges: not sharing vectorizer, might be a deoptimization

B. Vectorize in LLVM optimizations (LoopVectorizer)
   - Chapel front-end generates loops with parallel_loop_access
   - Challenges: user-defined reductions, querying vector lane

Vectorizing in the Front-End

- It would be lower level than most front end operations
  - a lot depends on the processor:
    - vector width
    - supported vector operations
    - whether or not vectorizing is profitable
- Front-end vectorization reasonable if details are simple
- Presumably there is a reason that vectorization normally runs late in the LLVM optimization pipeline...
- Misses out on a chance to share with the community

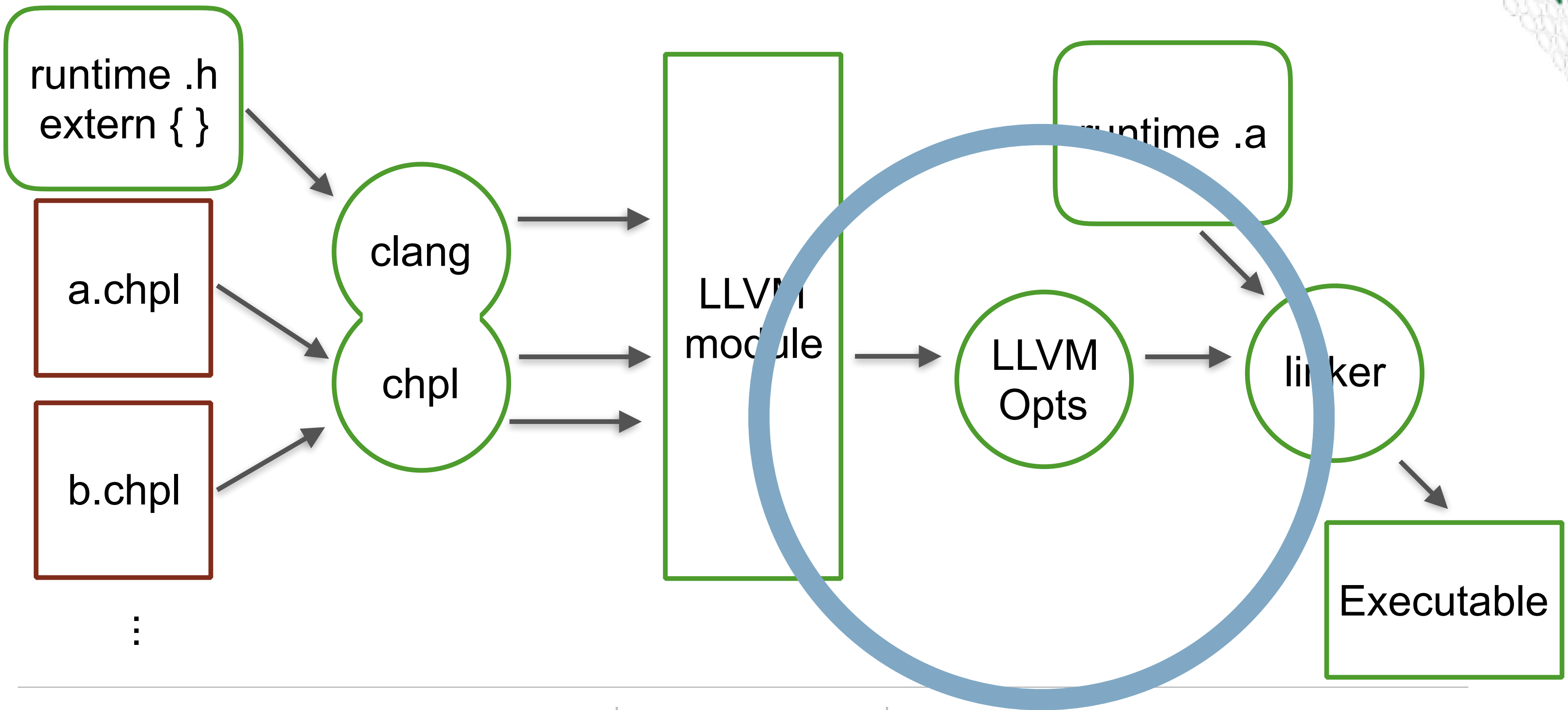
Vectorizing in as an LLVM optimization

- parallel_loop_access good for most of loop body...
- ... but what about user-defined reductions?
- Would opaque function calls help?
  - front-end generates opaque_accumulate function calls
  - after vectorization, these are replaced with real reduction ops
  - vectorizer would see something like this:

```
define opaque_accumulate(...) readnone
for ... {
   load/stores with parallel_loop_access ...
   %acc = opaque_accumulate(%acc, %value)
}
```

  - Would it interfere too much with the vectorizer? Harm cost modelling?
- Does LLVM need a canonical way to express custom reductions?

Communication Optimization in LLVM



Aside: Introducing PGAS Communication

Parallelism and Locality: Distinct in Chapel

- This is a parallel, but local program:

```
coforall i in 1..msgs do
  writeln("Hello from task ", i);
```

- This is a distributed, but serial program:

```
writeln("Hello from locale 0!");
on Locales[1] do writeln("Hello from locale 1!");
on Locales[2] do writeln("Hello from locale 2!");
```

- This is a distributed parallel program:

```
coforall i in 1..msgs do
  on Locales[i%numLocales] do
    writeln("Hello from task ", i,
            " running on locale ", here.id);
```

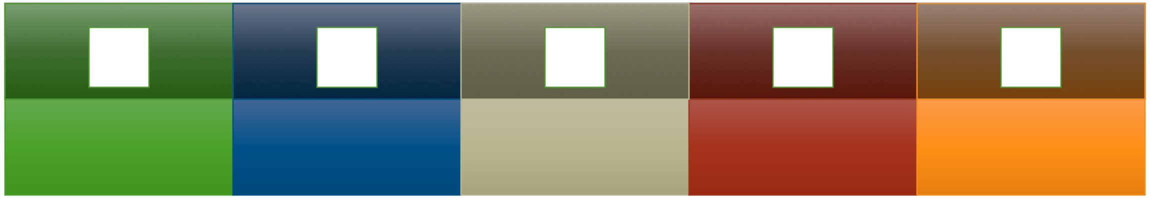
Partitioned Global Address Space (PGAS) Languages
(Or more accurately: partitioned global namespace languages)

- abstract concept:
  - support a shared namespace on distributed memory
    - permit parallel tasks to access remote variables by naming them
  - establish a strong sense of ownership
    - every variable has a well-defined location
    - local variables are cheaper to access than remote ones
- traditional PGAS languages have been SPMD in nature
  - best-known examples: Fortran 2008's co-arrays, Unified Parallel C (UPC)

SPMD PGAS Languages (using a pseudo-language, not Chapel)

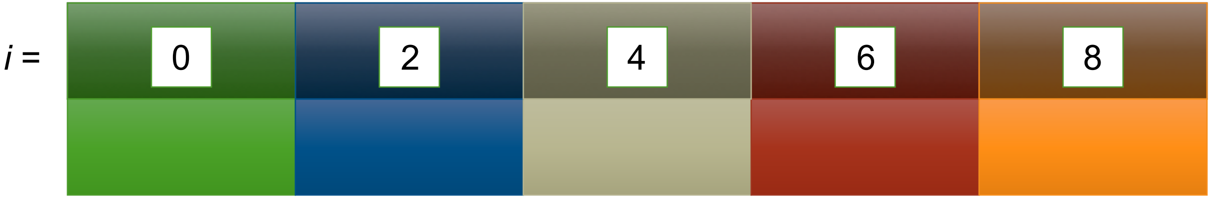
```
shared int i(*);            // declare a shared variable i
```

i =



SPMD PGAS Languages (using a pseudo-language, not Chapel)

```
shared int i(*);             // declare a shared variable i
function main() {
  i = 2*this_image();        // each image initializes its copy
```



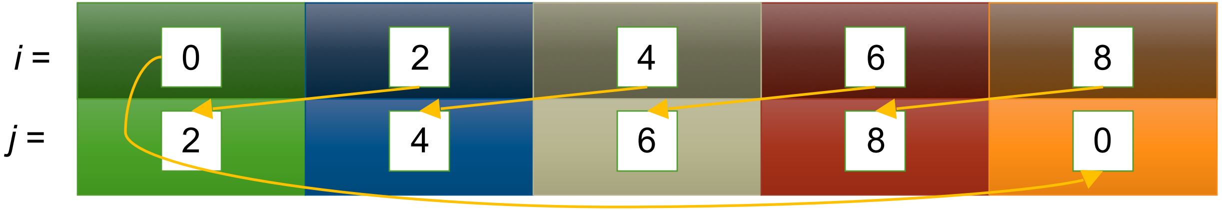
SPMD PGAS Languages (using a pseudo-language, not Chapel)

```
shared int i(*);                 // declare a shared variable i
function main() {
  i = 2*this_image();            // each image initializes its copy

  private int j;                 // declare a private variable j
```

i =	0	2	4	6	8
j =					

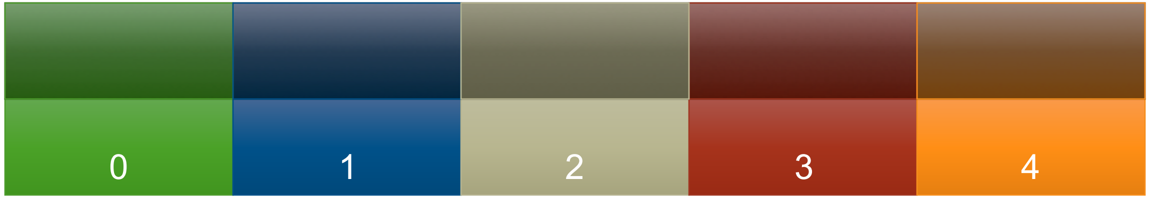
SPMD PGAS Languages (using a pseudo-language, not Chapel)

```
shared int i(*);            // declare a shared variable i
function main() {
  i = 2*this_image();       // each image initializes its copy
  barrier();
  private int j;            // declare a private variable j
  j = i( (this_image()+1) % num_images() );
    // ^^ access our neighbor's copy of i; compiler and runtime implement the communication
    // Q: How did we know our neighbor had an i?
    // A: Because it's SPMD – we're all running the same program so if we have an i, so do they.
```



Chapel and PGAS

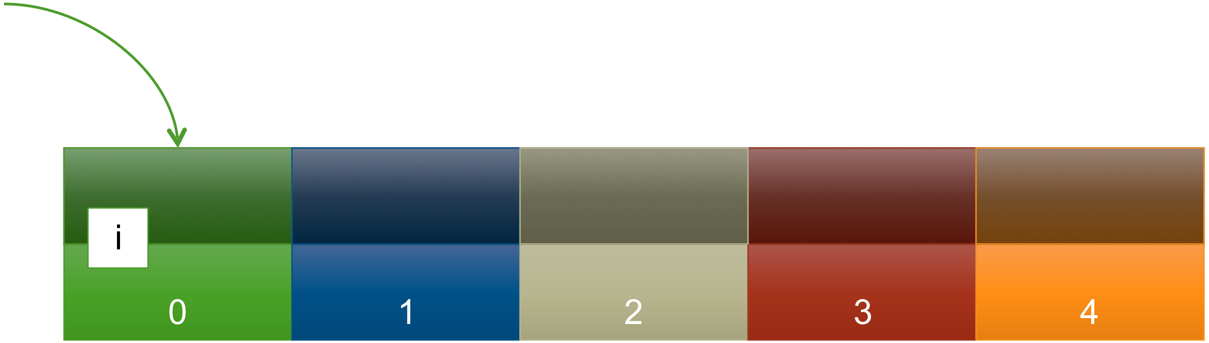
- Chapel is PGAS, but unlike most, it's not inherently SPMD
  - never think about "the other copies of the program"
  - "global name/address space" comes from lexical scoping
    - as in traditional languages, each declaration yields one variable
    - variables are stored on the locale where the task declaring it is executing



Locales (think: "compute nodes")

Chapel: Scoping and Locality

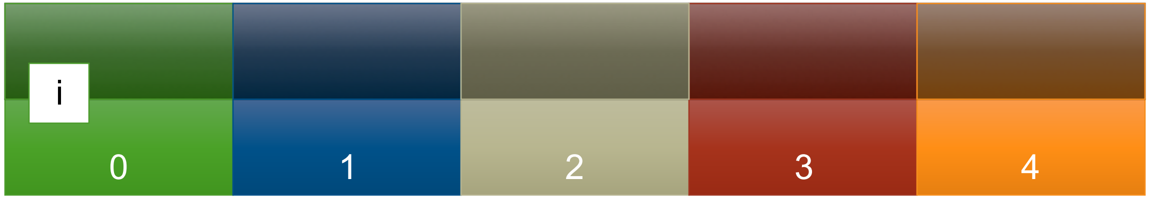
```
var i: int;
```



Locales (think: “compute nodes”)

Chapel: Scoping and Locality

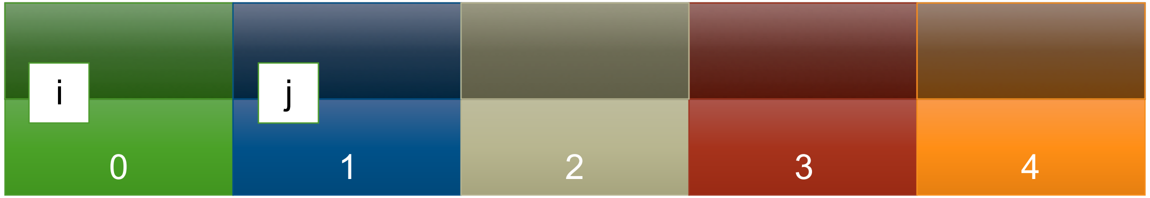
```
var i: int;
on Locales[1] {
```



Locales (think: “compute nodes”)

Chapel: Scoping and Locality

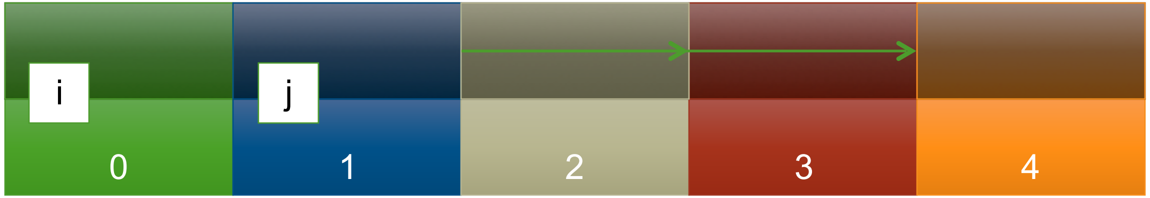
```
var i: int;
on Locales[1] {
  var j: int;
```



Locales (think: “compute nodes”)

Chapel: Scoping and Locality

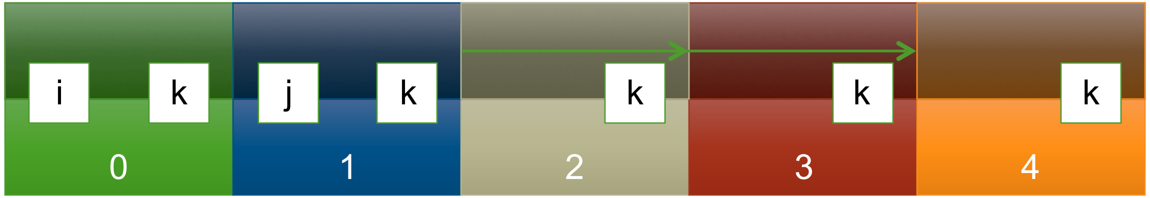
```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
```



Locales (think: "compute nodes")

Chapel: Scoping and Locality

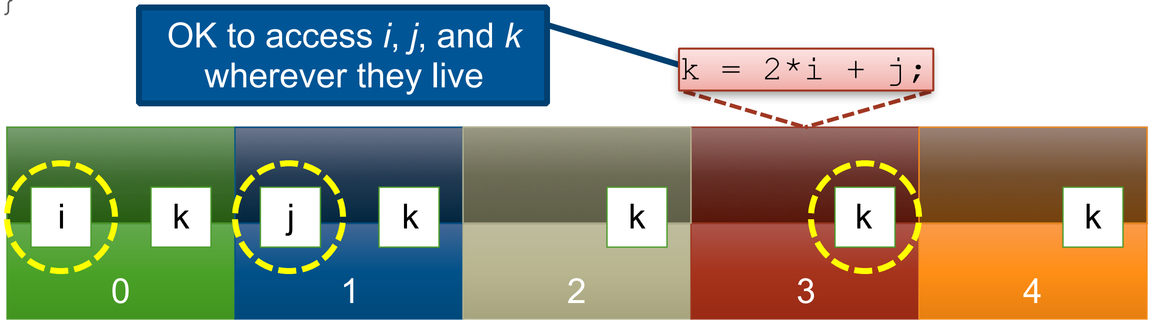
```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k: int;
      …
    }
  }
}
```



Locales (think: "compute nodes")

Chapel: Scoping and Locality

```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k: int;
      k = 2*i + j;
    }
  }
}
```

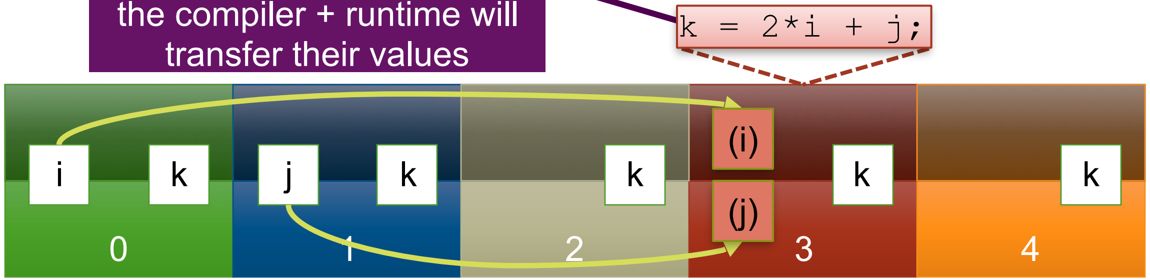


Locales (think: "compute nodes")

Chapel: Scoping and Locality

```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k: int;
      k = 2*i + j;
    }
  }
}
```

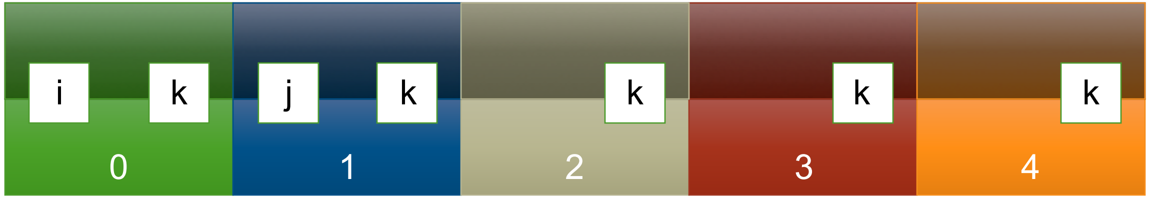
here, *i* and *j* are remote, so the compiler + runtime will transfer their values



Locales (think: “compute nodes”)

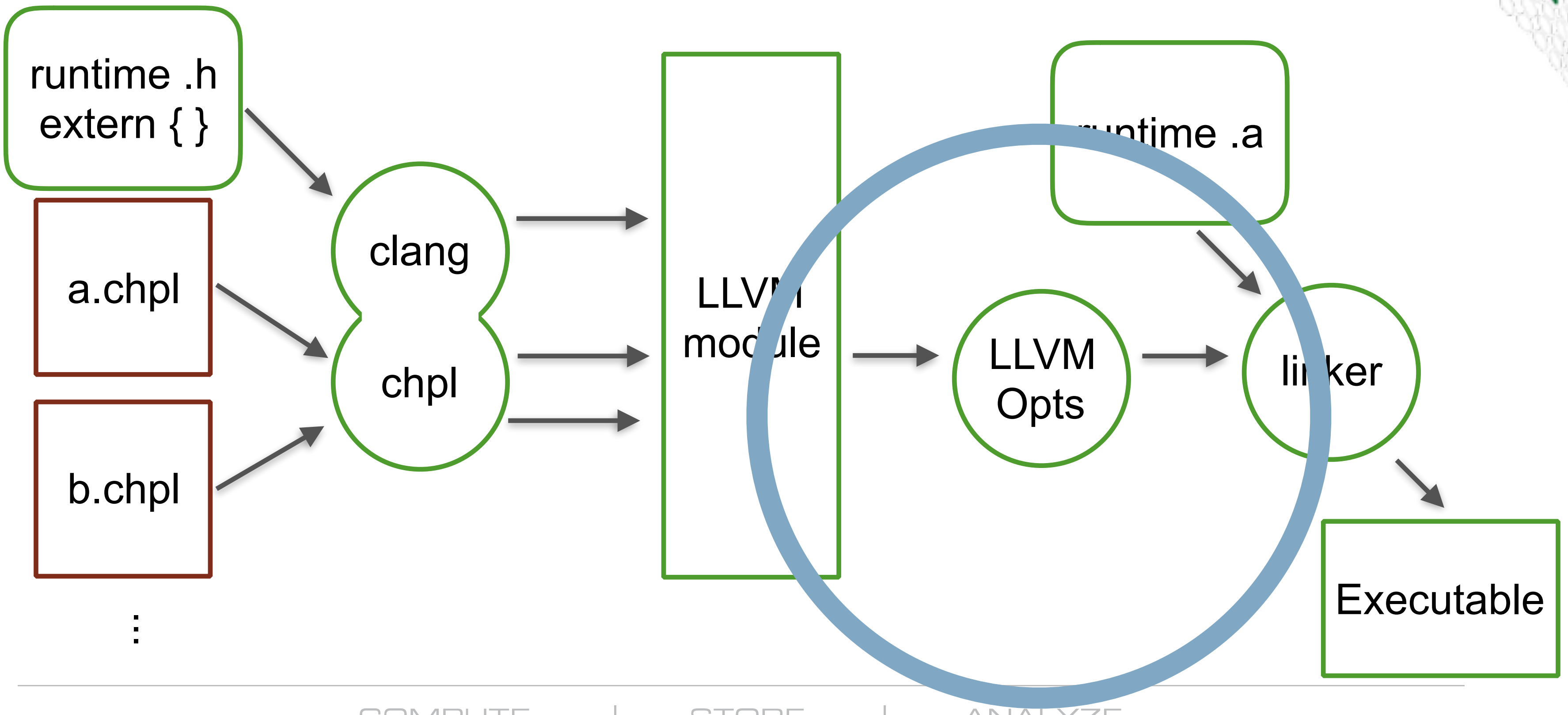
Chapel: Locality queries

```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k: int;

      ...here...          // query the locale on which this task is running
     ...j.locale...       // query the locale on which j is stored
    }
  }
}
```



Locales (think: "compute nodes")

Communication Optimization in LLVM



Communication Optimization: Overview

- Idea is to use LLVM passes to optimize GET and PUT
- Enabled with --llvm-wide-opt compiler flag
- First appeared in Chapel 1.8
- Unfortunately was not working in 1.15 and 1.16 releases

Communication Optimization: In a Picture

```
// 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

```
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
```

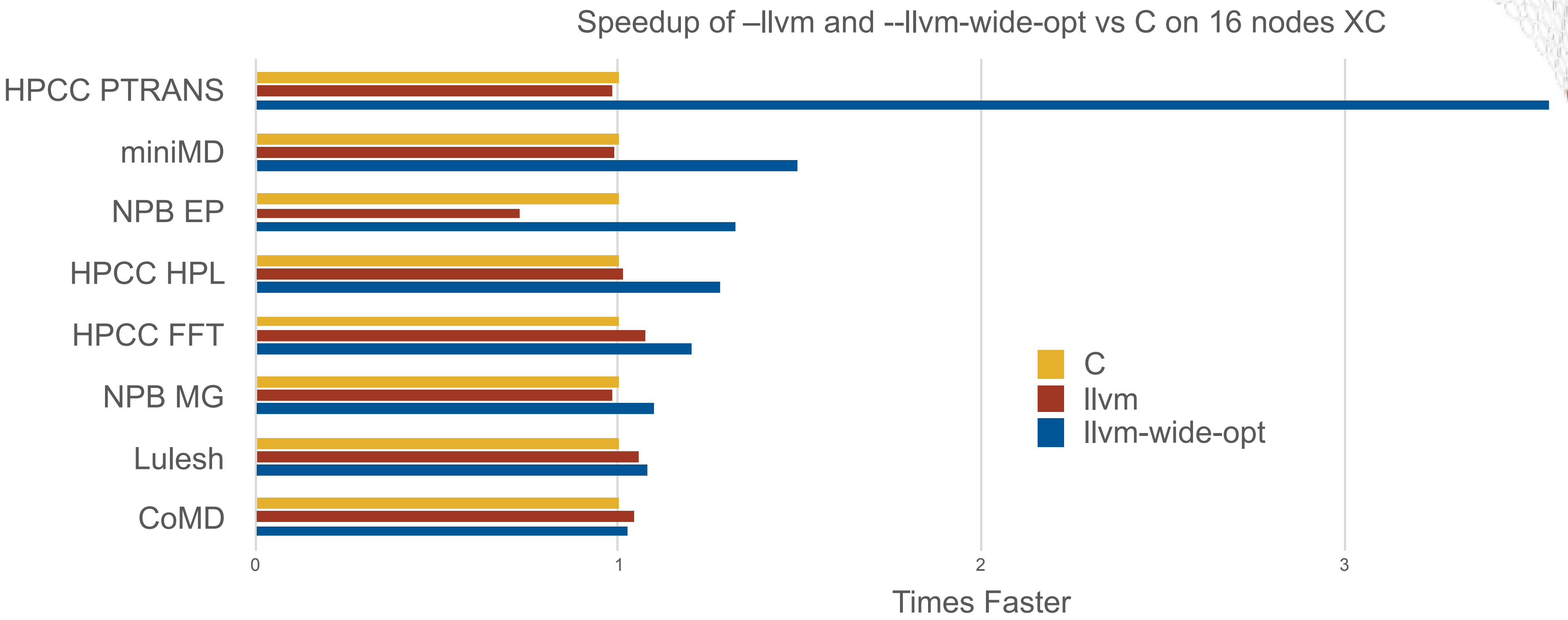
Communication Optimization: Details

- Uses existing LLVM passes to optimize GET and PUT
  - GET/PUT represented as load/store with special pointer type
  - normal LLVM optimizations run and optimize load/store as usual
  - a custom LLVM pass lowers them back to calls to the Chapel runtime
- Optimization gains from this strategy can be significant
  - See "LLVM-based Communication Optimizations for PGAS Programs"
- Historically, needed packed wide pointers as workaround
  - wide pointer normally stored as a 128-bit struct: {node id, address}
  - bugs in LLVM 3.3 prevented using 128-bit pointers
  - packed wide pointers store node id in high bits of a 64-bit address
  - led to scalability constraints - maximum of 65536 nodes
  - sometimes made --llvm-wide-opt code slower than C backend

Communication Optimization: Recent Work

- Fixed --llvm-wide-opt
- Removed reliance on packed wide pointers
- Now generates LLVM IR with 128-bit pointers
  - revealed (only) a few LLVM bugs (so far)
    - BasicAA - needed to use APInt more (not just int64_t)
    - ValueTracking - error using APInt

Comm Opt: Impact



Future Work

- Chapel 1.17 - hope to make --llvm the default
- Migrate some Chapel-specific optimizations to LLVM
- Continue improving the LLVM IR that Chapel generates
- Separate compilation & link-time optimization
- Chapel interpreter using LLVM JIT
- Using a shared parallel LLVM IR

Thanks

- Thanks for your
  - attention
  - great discussions
  - patch reviews
  - related work!

- Check us out at https://chapel-lang.org

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, and URIKA. 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.

Backup Slides

Chapel Community R&D Efforts

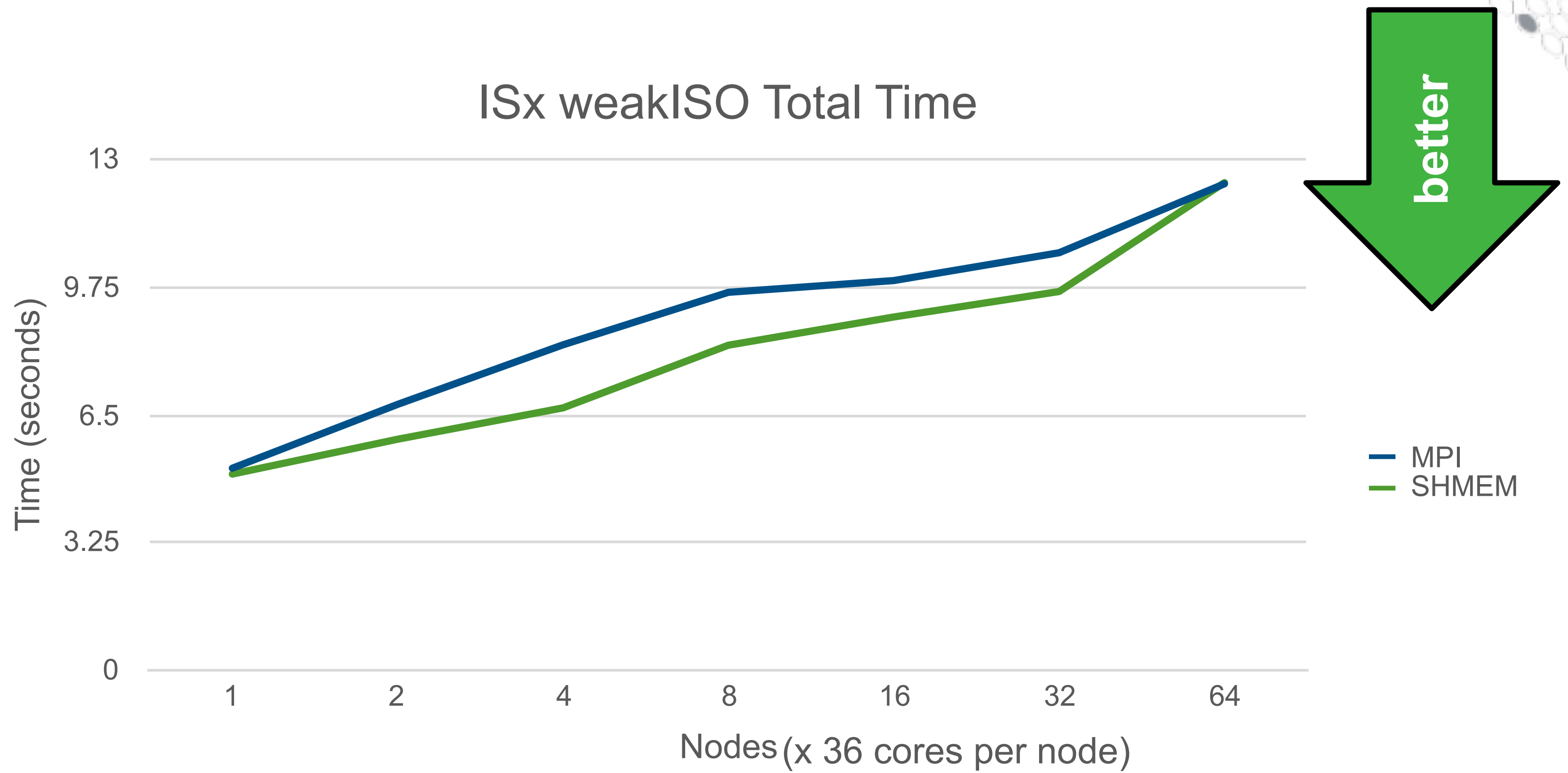
(and several others…)

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

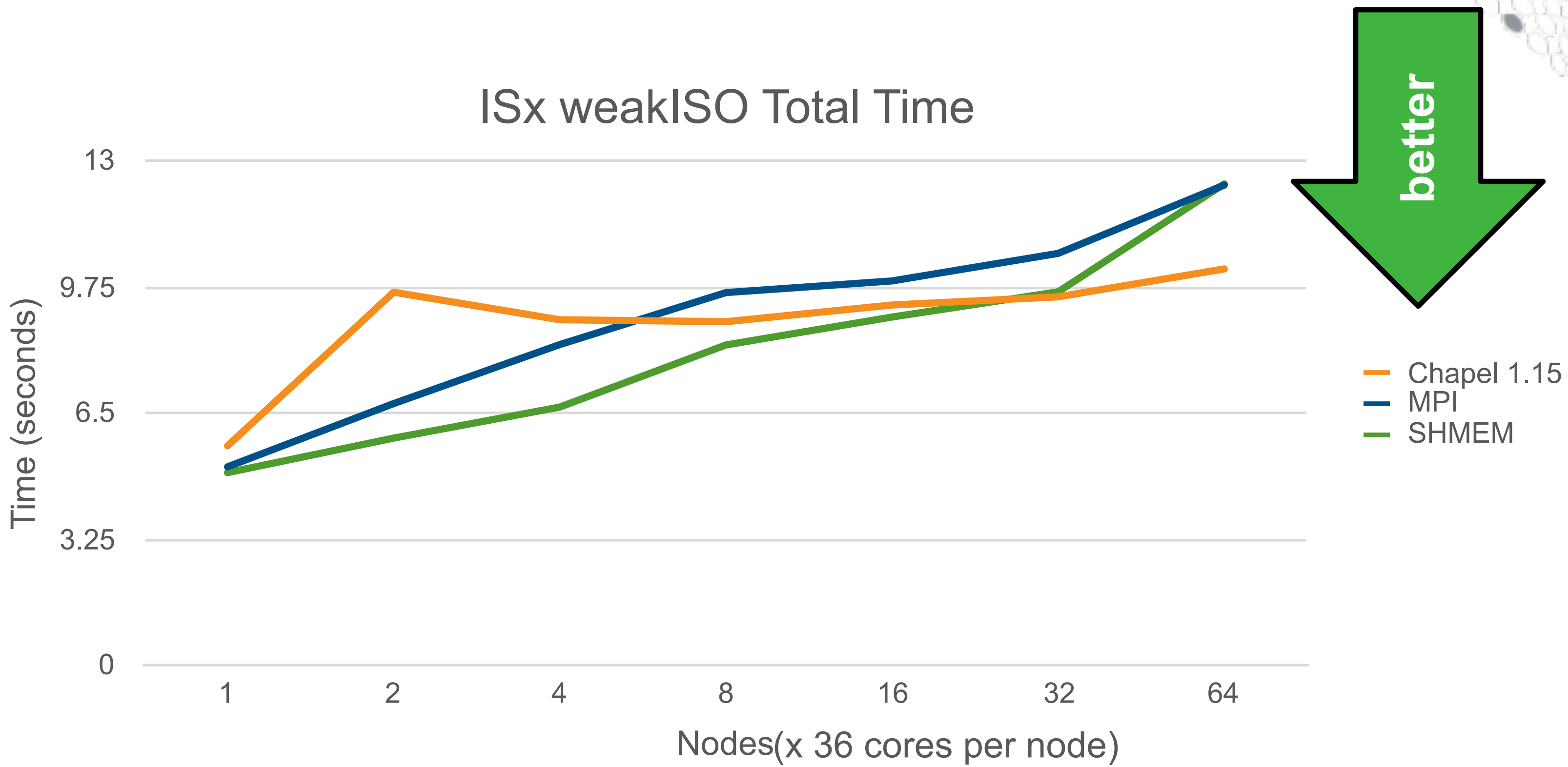
Contributions to LLVM/clang

- Add clang CodeGen support for generating field access
  - supports 'extern record'
  - https://reviews.llvm.org/D38473
- Fix a bug in BasicAA – crashing with 128-bit pointers
  - enables --llvm-wide-opt with {node, address} wide pointers
  - https://reviews.llvm.org/D38499
- Fix a bug in ValueTracking – crashing with 128-bit pointers
  - enables --llvm-wide-opt with {node, address} wide pointers
  - https://reviews.llvm.org/D38501

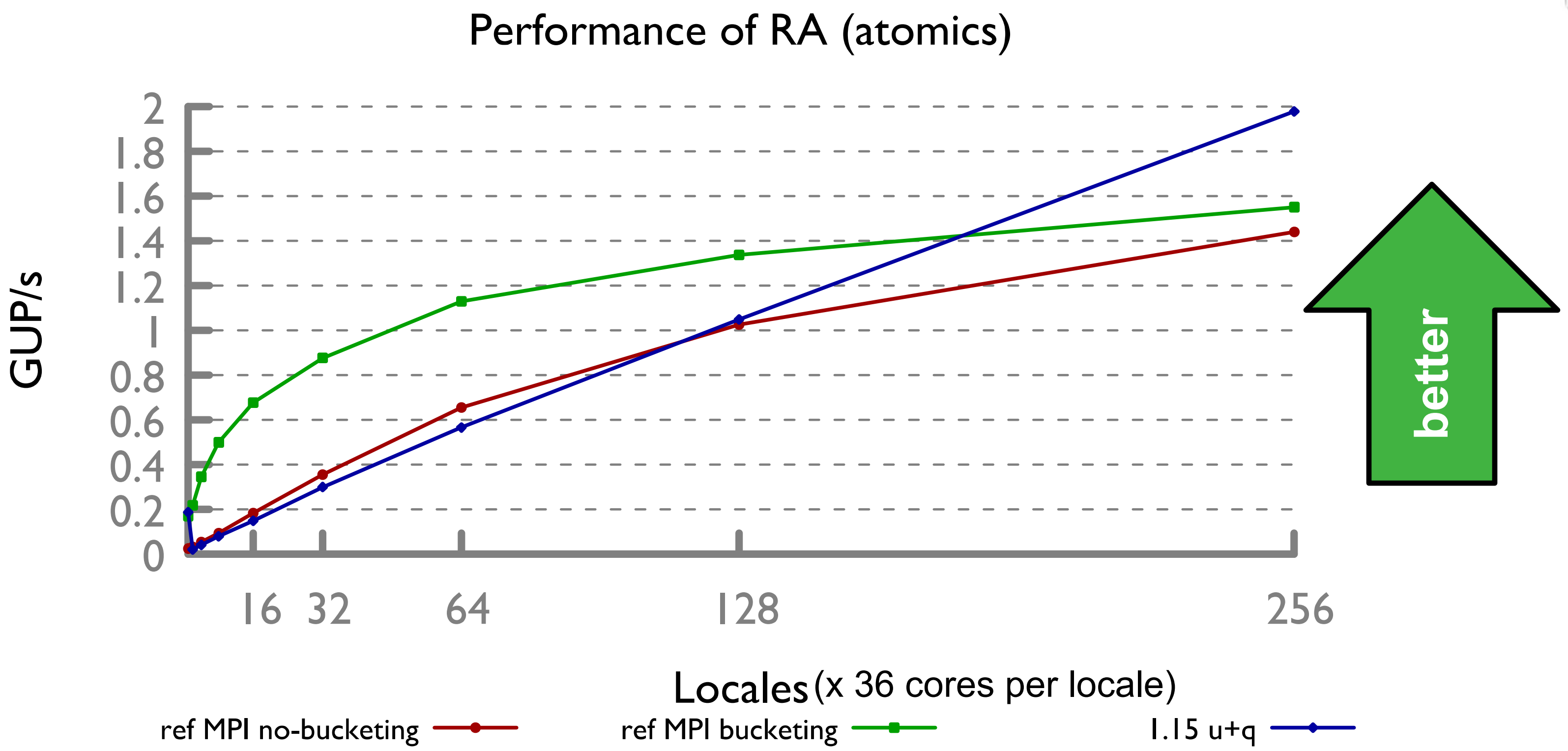
ISx Execution Time: MPI, SHMEM



ISx Execution Time: MPI, SHMEM, Chapel



RA Performance: Chapel vs. MPI

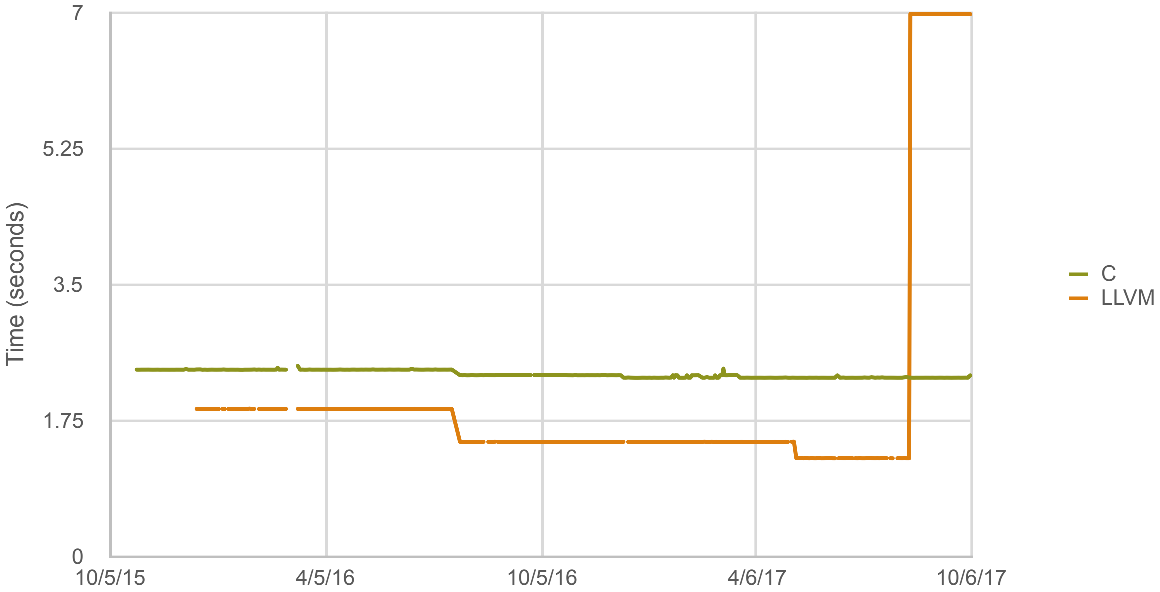


Chapel+LLVM - Google Summer of Code

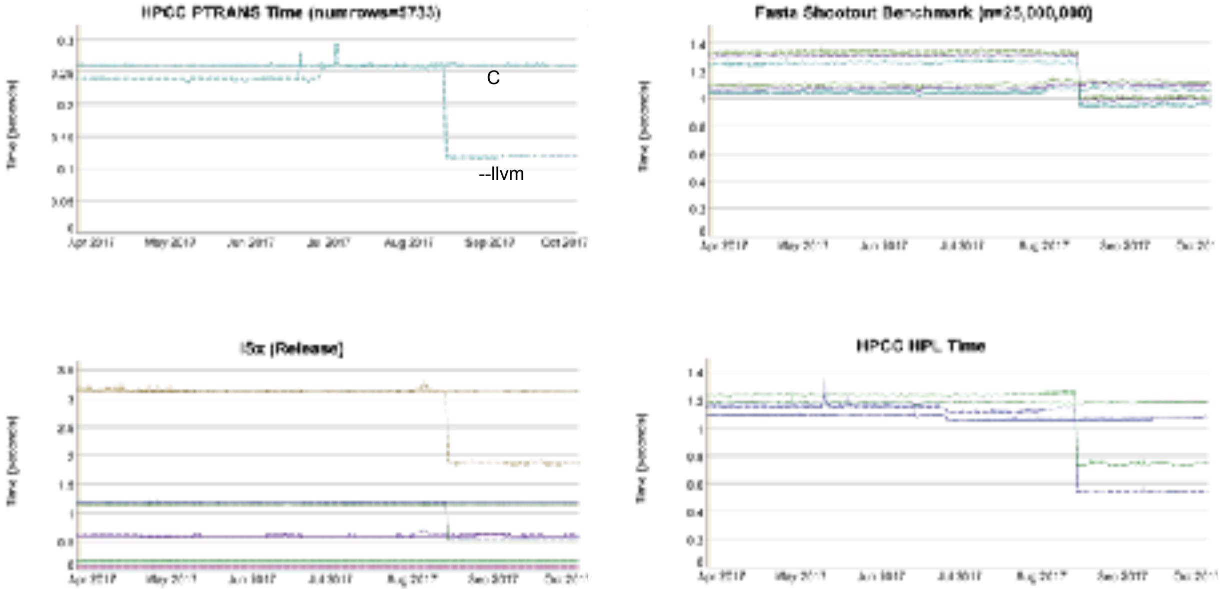
- Przemysław Leśniak contributed many improvements:
  - mark signed integer arithmetic with 'nsw' to improve loop optimization
  - command-line flags to emit LLVM IR at particular points in compilation
  - new tests that use LLVM tool FileCheck to verify emitted LLVM IR
  - mark order-independent loops with llvm.parallel_loop_access metadata
  - mark const variables with llvm.invariant.start
  - enable LLVM floating point optimization when --no-ieee-float is used
  - add nonnull attribute to ref arguments to functions
  - add a header implementing clang built-ins to improve 'complex' performance

Performance Regression: LLVM 3.7 to 4 upgrade

LCALS find_first_min went from 2x faster than C to 3x slower



Performance Improvements: LLVM 3.7 to 4

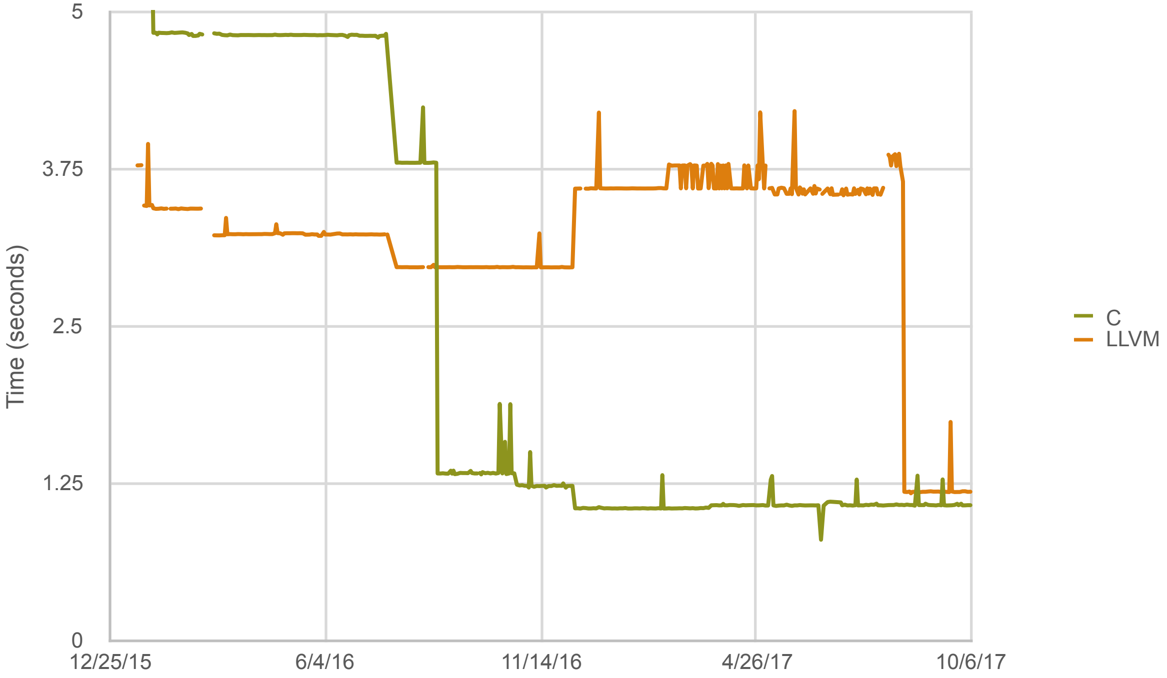


Performance Improvements: GSoC nsw

PRK stencil got 3x faster with no-signed-wrap on signed integer addition – loop induction variable now identified

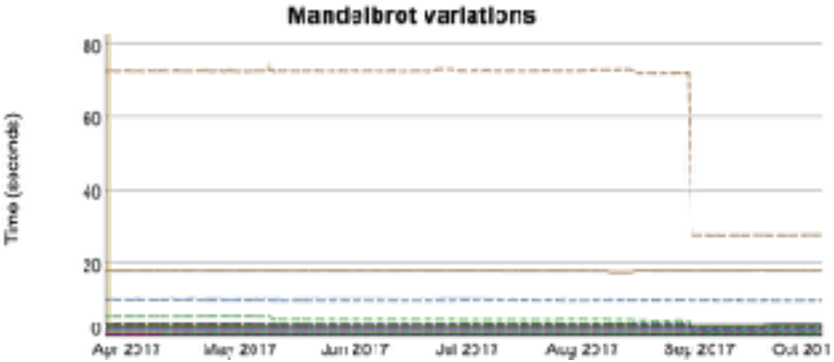
Performance Improvement: GSoC fast float

LCALS fir got 3x faster with LLVM floating point optimizations enabled for --no-ieee-float



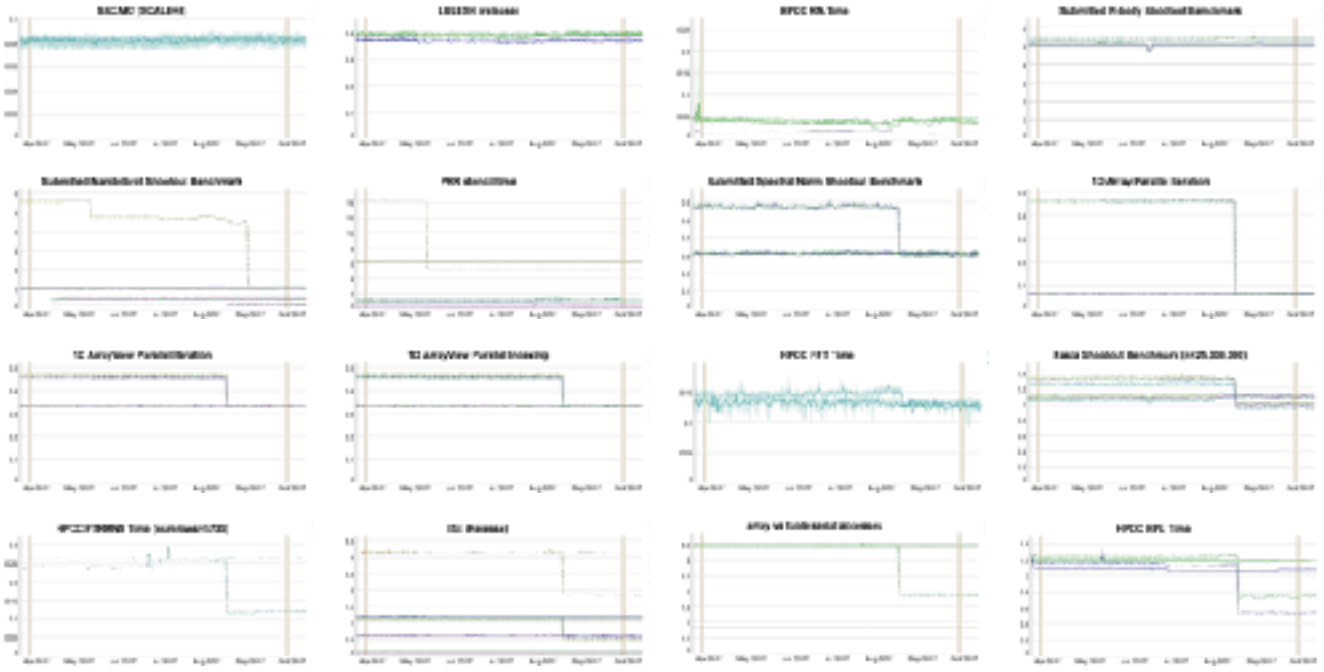
Performance Improvements: Built-ins Header

complex version of Mandelbrot got 3x faster with header implementing clang built-ins



--llvm is Now Competitive

- Benchmark runtime now competitive or better with --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, and URIKA. 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.