Chapel's Language-based Approach to Performance Portability

Brad Chamberlain

SIAM CSE19, MS95: Performance Portability and Numerical Libraries

February 25, 2019

bradc@cray.com
chapel-lang.org
@ChapelLanguage

Performance Portability: The Dream

Performance Portability: when software performs well across a range of architectures and problem configurations with modest development and maintenance effort

Performance Portability: The Harsh Reality

Whenever system architectures expose a unique feature…

For example:

- vector instructions
- accelerators
- special flavors of memory
- RDMA (Remote Direct Memory Access)
- network support for atomic operations

…performance portability becomes challenging

- **Use the feature?**
- **Ignore it?**

Performance Portability: The Harsh Reality

Whenever system architectures expose a unique feature…

For example:

- vector instructions
- accelerators
- special flavors of memory
- RDMA (Remote Direct Memory Access)
- network support for atomic operations

…performance portability becomes challenging

- **Use the feature?** ⇒ will likely break performance portability to other systems
- **Ignore it?**

Performance Portability: The Harsh Reality

Whenever system architectures expose a unique feature…

For example:

- vector instructions
- accelerators
- special flavors of memory
- RDMA (Remote Direct Memory Access)
- network support for atomic operations

…performance portability becomes challenging

- **Use the feature?** ⇒ will likely break performance portability to other systems
- **Ignore it?** ⇒ leaves performance on the table, wasting resources

Performance Portability: The Harsh Reality

Whenever system architectures expose a unique feature…

For example:

- vector instructions
- accelerators
- special flavors of memory
- RDMA (Remote Direct Memory Access)
- network support for atomic operations

…performance portability becomes challenging

- **Use the feature?** ⇒ will likely break performance portability to other systems
- **Ignore it?** ⇒ leaves performance on the table, wasting resources
- **Support multiple implementations?** ⇒ lots of code engineering and upkeep

Performance Portability: The Harsh Reality

Whenever system architectures expose a unique feature…

For example:

- vector instructions
- accelerators
- special flavors of memory
- RDMA (Remote Direct Memory Access)
- network support for atomic operations

…performance portability becomes challenging

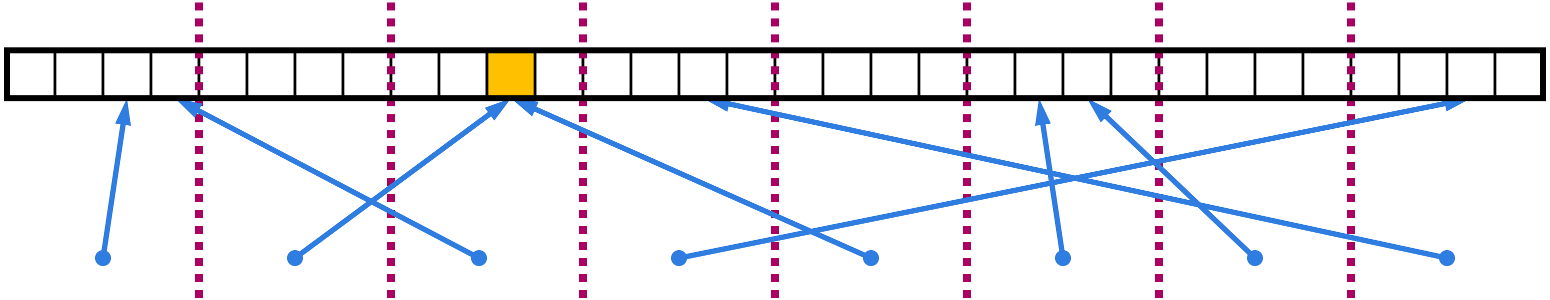
- **Use the feature?** ⇒ will likely break performance portability to other systems
- **Ignore it?** ⇒ leaves performance on the table, wasting resources
- **Support multiple implementations?** ⇒ lots of code engineering and upkeep

HPCC RA

An illustrative example

Case Study: HPCC Random Access (RA)

Data Structure: distributed table



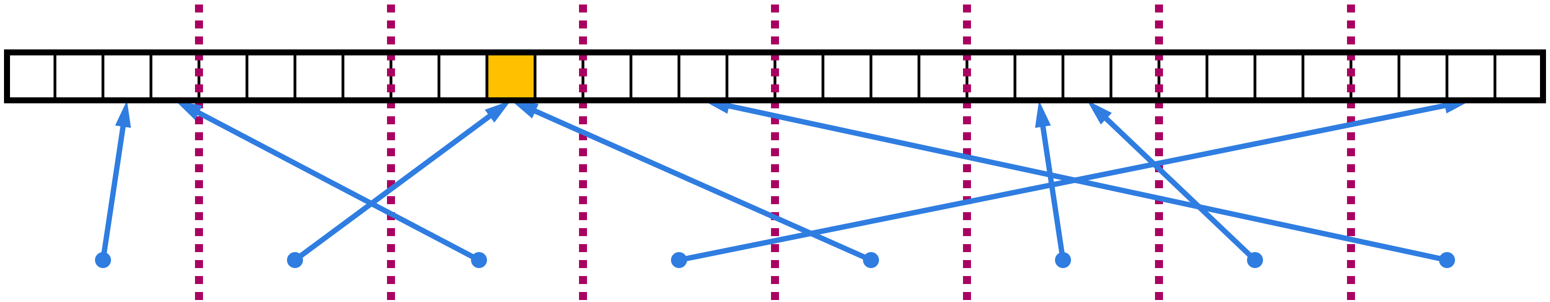
Computation: update random table locations in parallel

Two variations:

- **lossless:** don't allow any updates to be lost
- **lossy:** permit some fraction of updates to be lost

Case Study: HPCC Random Access (RA)

Data Structure: distributed table



Computation: update random table locations in parallel

Two variations:

- **lossless:** don't allow any updates to be lost
- **lossy:** permit some fraction of updates to be lost

HPCC RA (lossless): Pseudocode

parallel for *val* **in** *RandomValues*:

  $loc \leftarrow val \ \& \ mask$

  $Table[loc] \leftarrow Table[loc]$ **atomic-xor** *val*

HPCC RA (lossless): Pseudocode

parallel for *val* **in** *RandomValues*:

    loc ← *val* & *mask*

    Table[*loc*] ← *Table*[*loc*] **atomic-xor** *val*

HPCC RA: From pseudocode to conventional code

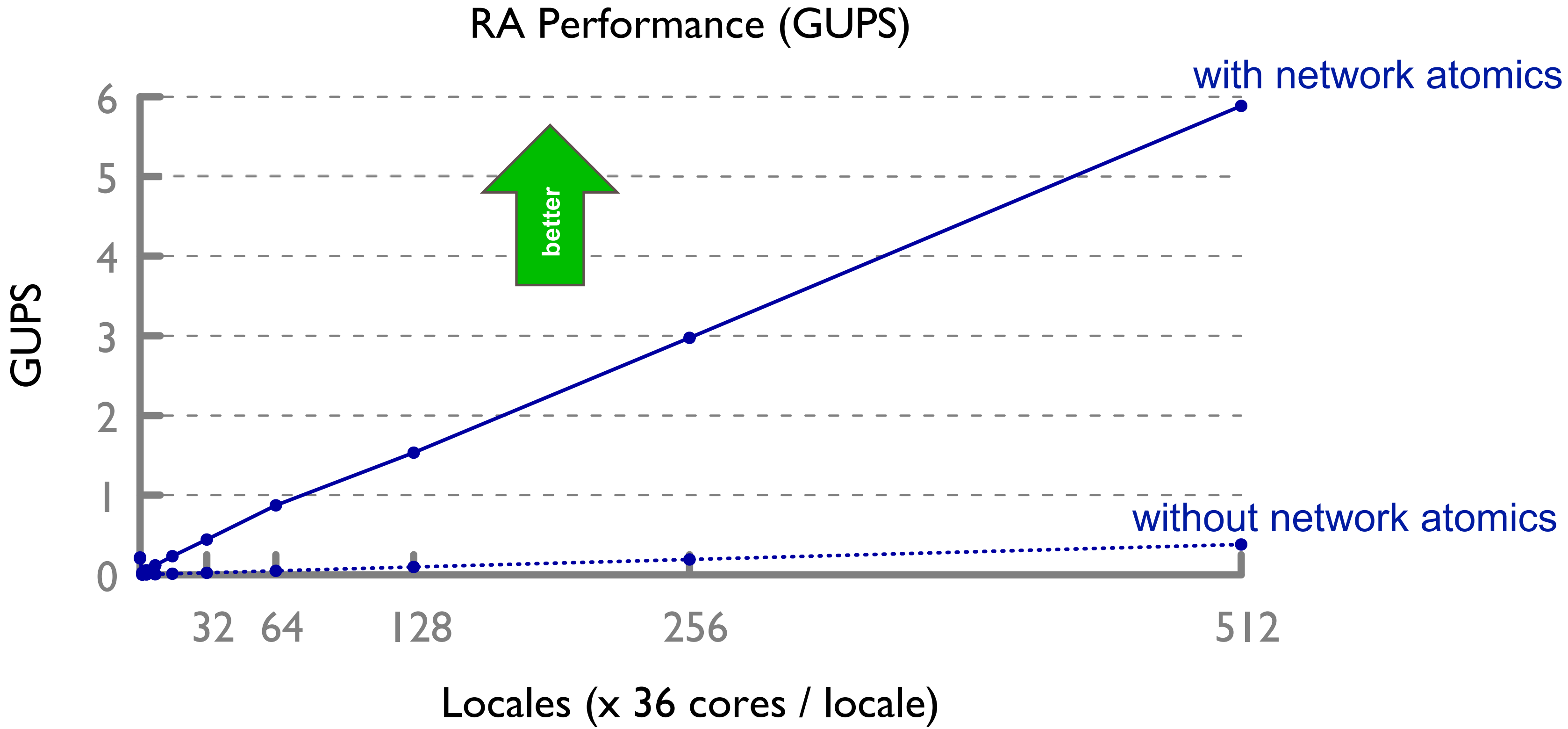
With network atomics:

- use a vendor-specific networking library
  - e.g., uGNI
- use a portable library supporting network atomics
  - e.g., GASNet-EX, OpenSHMEM, OFI (libfabric)

Without network atomics:

- use active messages + processor atomics
  - e.g., GASNet-EX + C11 atomics

HPCC RA: with or without network atomics



HPCC RA: From pseudocode to conventional code

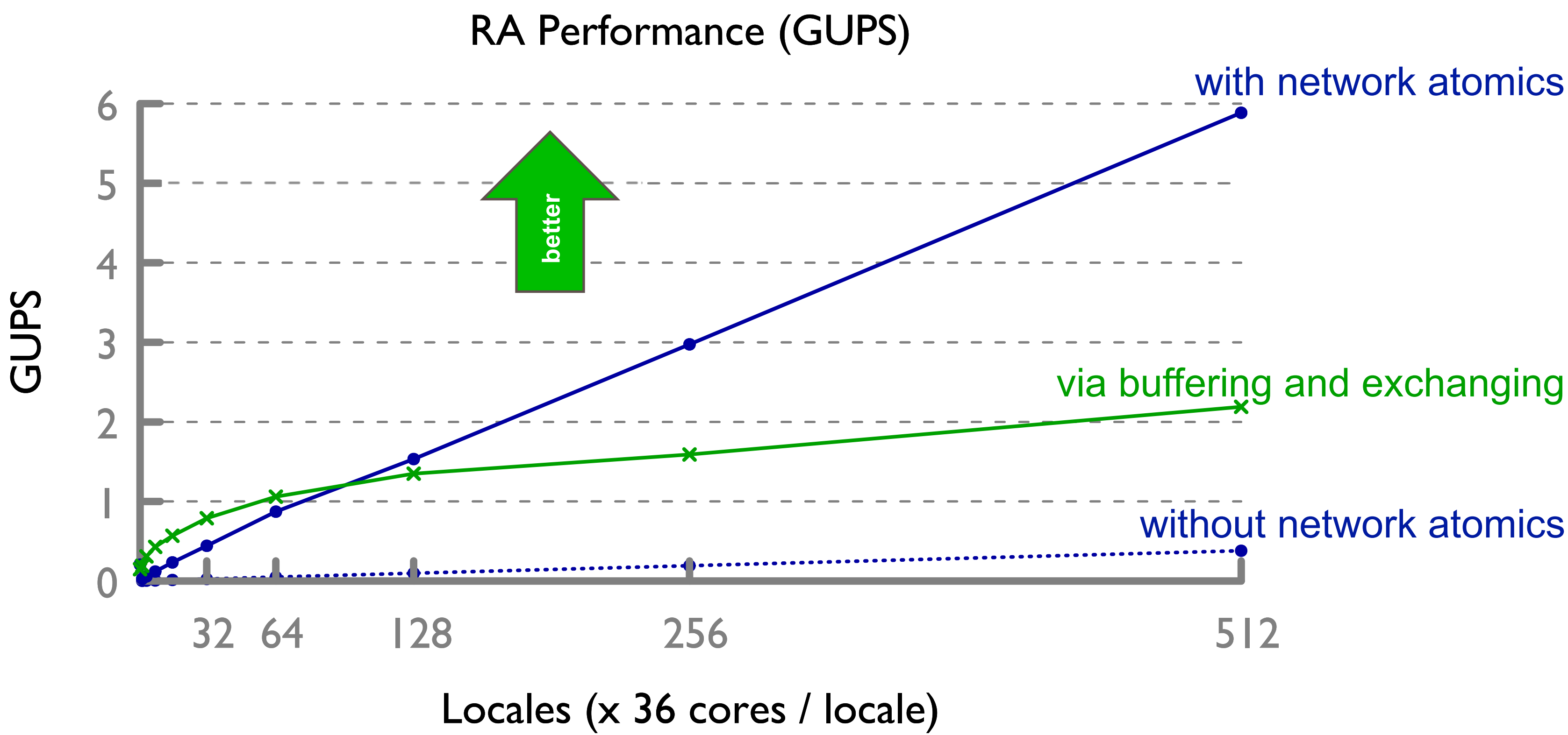
With network atomics:

- use a vendor-specific networking library
  - e.g., uGNI
- use a portable library supporting network atomics
  - e.g., GASNet-EX, OpenSHMEM, OFI (libfabric)

Without network atomics:

- use active messages + processor atomics
  - e.g., GASNet-EX + C11 atomics
- buffer updates locally, exchange buffers, and compute (a switch in algorithm)
  - e.g., MPI

HPCC RA: buffering vs. network atomics



The Case for Languages

A Historical Look at Performance Portability

1950's: Period of rapid hardware evolution and diversity

- performance coding was done in assembly / machine code
  - ⇒ by definition, a lack of performance portability
- FORTRAN was invented to help with this challenge
  - users were initially skeptical that it would perform well enough
  - ultimately, won over by productivity benefits and optimizing compilers

Since then: other high-level languages have followed suit for other domains

- e.g., C, C++, Java, Swift, …

Meanwhile, in present-day HPC…

- we're also experiencing a rapid evolution in hardware diversity
- we're programming via libraries, pragmas, DSLs (domain-specific languages), …
  - e.g., C/C++/Fortran + MPI + OpenMP / CUDA / OpenCL / Kokkos / … + …
  - obtaining good performance and scalability
  - but hitting performance portability challenges
    - by embedding architecture-specific assumptions
    - or by working hard to avoid them
- analogous to assembly language programming for specific HW/SW parallelism

Could programming languages help HPC programmers?

Why Consider New Languages at all?

Syntax	• High level, elegant syntax • Improve programmer productivity
Semantics	• Static analysis can help with correctness • We need a compiler (front-end)
Performance	• If optimizations are needed to get performance • We need a compiler (back-end)
Algorithms	• Language defines what is easy and hard • Influences algorithmic thinking

[Source: Kathy Yelick, CHIUW 2018 keynote: *Why Languages Matter More Than Ever*]

What is Chapel?

Chapel: A productive parallel programming language

- portable & scalable
- open-source & collaborative

Goals:

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

Chapel and Productivity

Chapel aims to be as…

…**programmable** as Python

…**fast** as Fortran

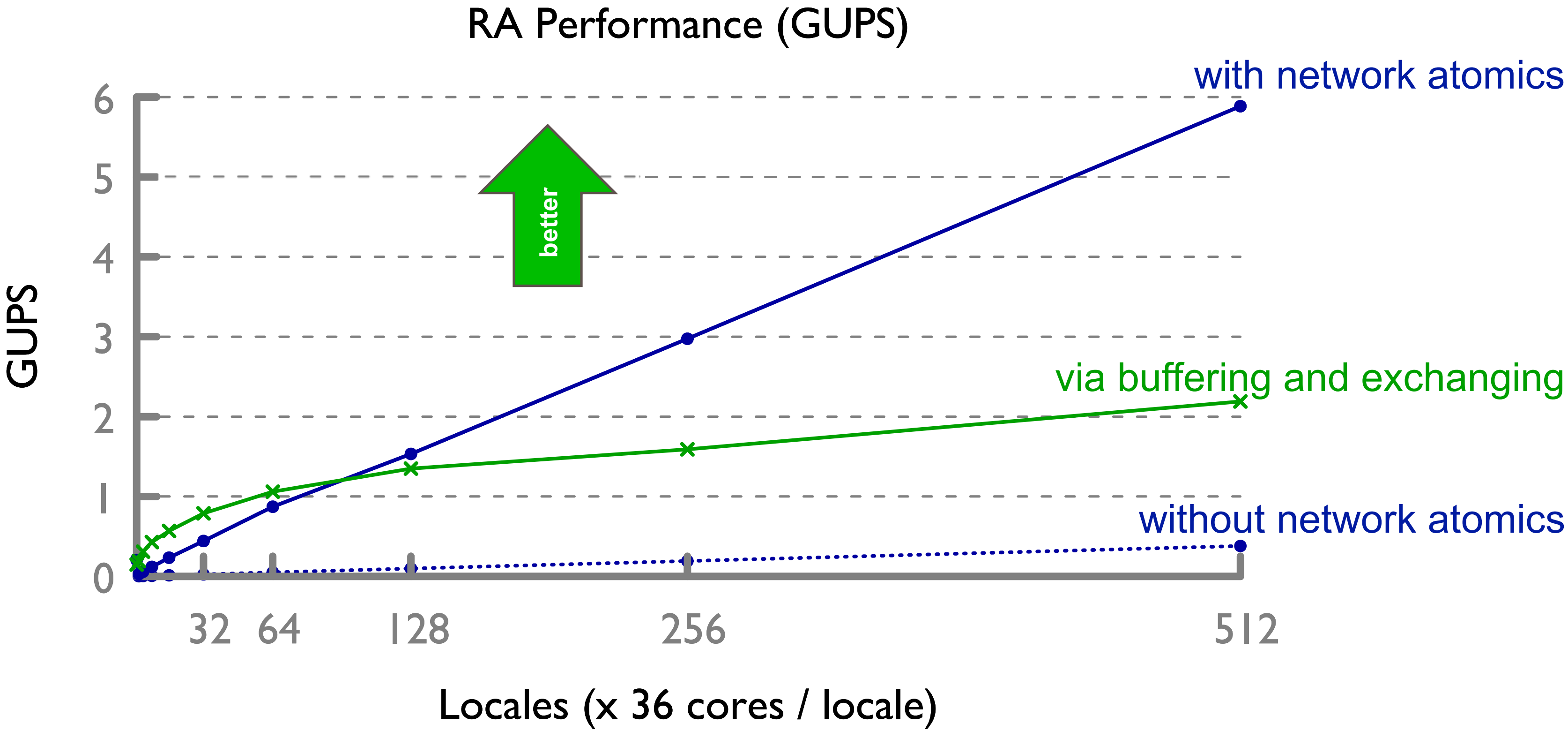
…**scalable** as MPI, SHMEM, or UPC

…**portable** as C

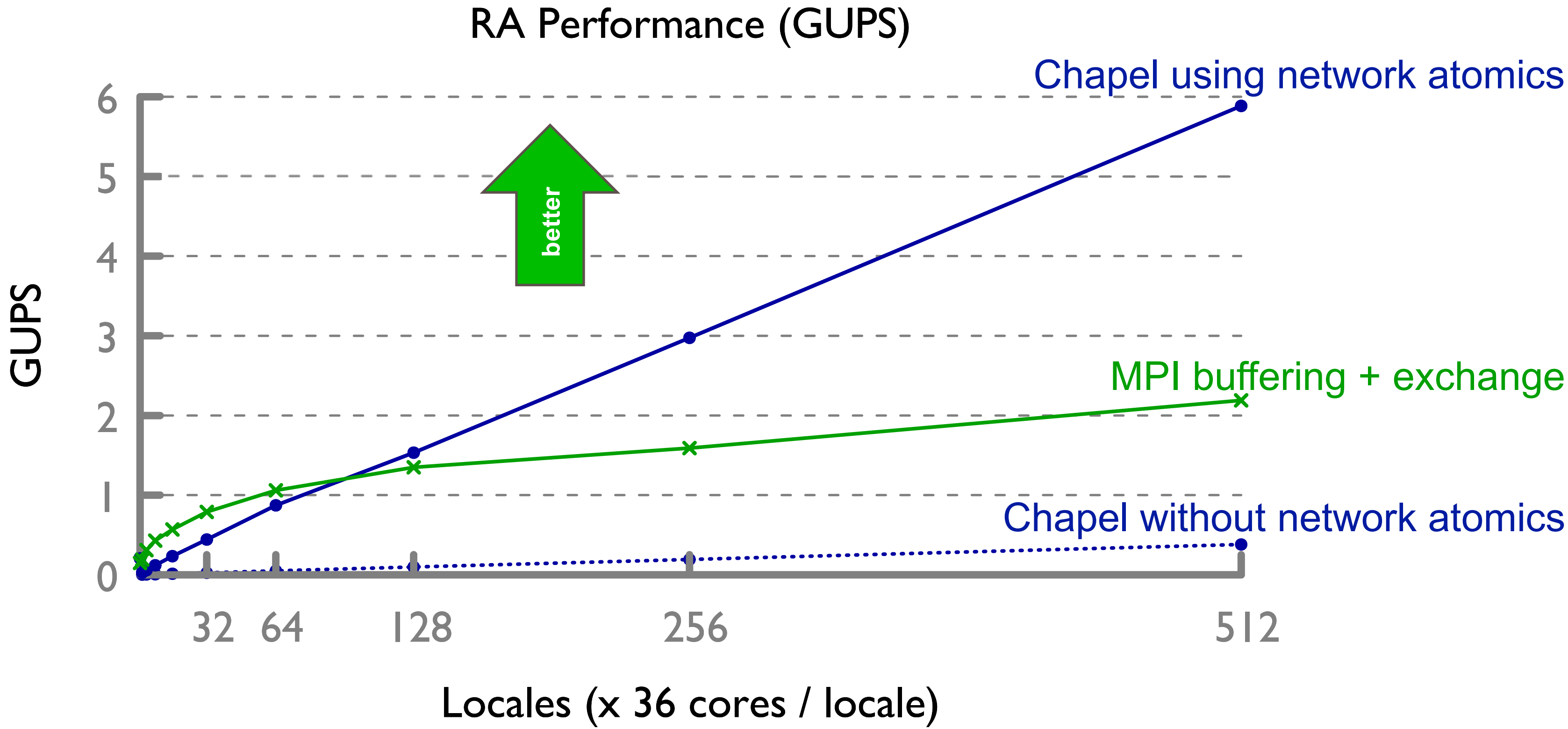
…**flexible** as C++

…**fun** as [your favorite programming language]

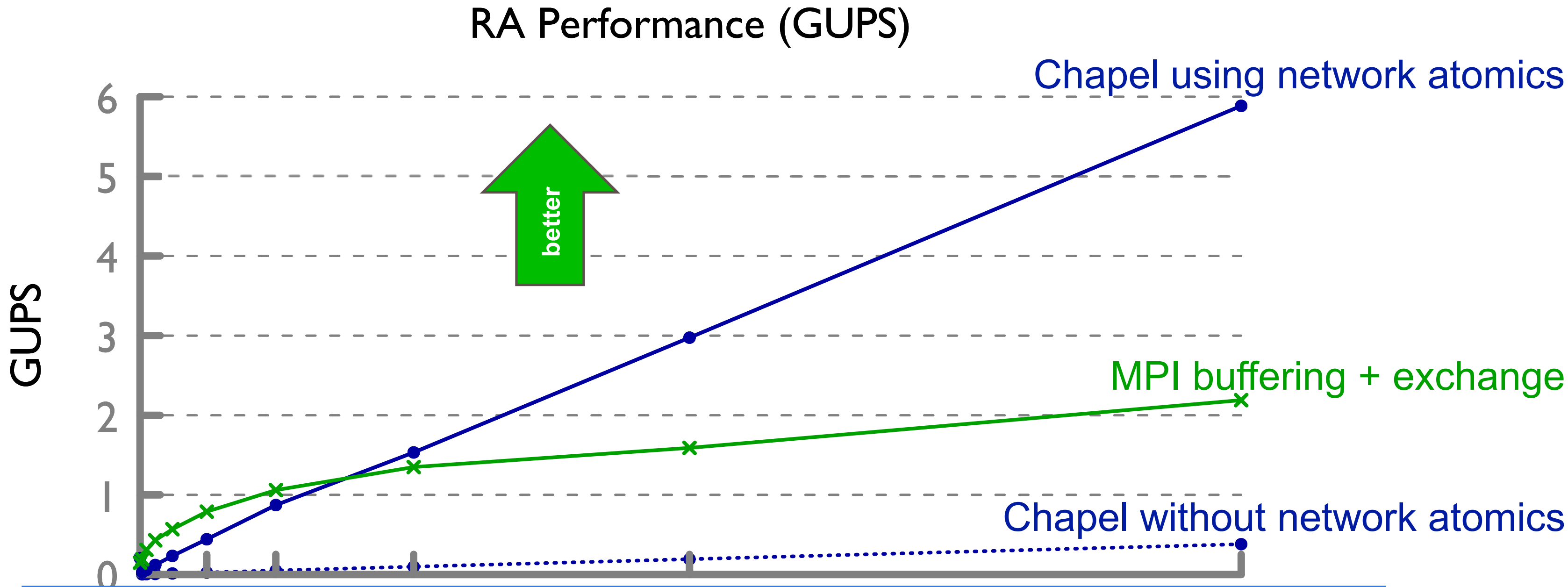
HPCC RA: buffering vs. network atomics



HPCC RA: MPI vs. Chapel



HPCC RA: MPI vs. Chapel



Cases like this in which a pair of programs perform asymmetrically relative to one another on systems with and without network atomics indicate a challenge to performance portability.

HPCC RA: MPI kernel

```
/* Perform updates to main table.  The scalar equivalent is:
 *
 *   for (i=0; i<NUPDATE; i++) {
 *     Ran = (Ran << 1) ^ (((s64Int) Ran < 0) ? POLY : 0);
 *     Table[Ran & (TABSIZE-1)] ^= Ran;
 *   }
 */

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
          MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
 while (i < SendCnt) {
   /* receive messages */
   do {
     MPI_Test(&inreq, &have_done, &status);
     if (have_done) {
       if (status.MPI_TAG == UPDATE_TAG) {
         MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
         bufferBase = 0;
         for (j=0; j < recvUpdates; j ++) {
           inmsg = LocalRecvBuffer[bufferBase+j];
           LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                          tparams.GlobalStartMyProc;
           HPCC_Table[LocalOffset] ^= inmsg;
         }
       } else if (status.MPI_TAG == FINISHED_TAG) {
         NumberReceiving--;
       } else
         MPI_Abort( MPI_COMM_WORLD, -1 );
       MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
                 MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
     }
   } while (have_done && NumberReceiving > 0);
   if (pendingUpdates < maxPendingUpdates) {
     Ran = (Ran << 1) ^ ((s64Int) Ran < ZERO64B ? POLY : ZERO64B);
     GlobalOffset = Ran & (tparams.TableSize-1);
     if ( GlobalOffset < tparams.Top)
       WhichPe = ( GlobalOffset / (tparams.MinLocalTableSize + 1) );
     else
       WhichPe = ( (GlobalOffset - tparams.Remainder) /
                   tparams.MinLocalTableSize );
     if (WhichPe == tparams.MyProc) {
       LocalOffset = (Ran & (tparams.TableSize - 1)) -
                      tparams.GlobalStartMyProc;
       HPCC_Table[LocalOffset] ^= Ran;
     }  else {
       HPCC_InsertUpdate(Ran, WhichPe, Buckets);
        pendingUpdates++;
     }
     i++;
   }
   else {
     MPI_Test(&outreq, &have_done, MPI_STATUS_IGNORE);
     if (have_done) {
       outreq = MPI_REQUEST_NULL;
       pe = HPCC_GetUpdates(Buckets, LocalSendBuffer, localBufferSize,
                            &peUpdates);
       MPI_Isend(&LocalSendBuffer, peUpdates, tparams.dtype64, (int)pe,
                 UPDATE_TAG, MPI_COMM_WORLD, &outreq);
       pendingUpdates -= peUpdates;
     }
   }
 }
/* send remaining updates in buckets */
while (pendingUpdates > 0) {
  /* receive messages */
  do {
    MPI_Test(&inreq, &have_done, &status);
    if (have_done) {
      if (status.MPI_TAG == UPDATE_TAG) {
        MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
        bufferBase = 0;
        for (j=0; j < recvUpdates; j ++) {
          inmsg = LocalRecvBuffer[bufferBase+j];
          LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                         tparams.GlobalStartMyProc;
          HPCC_Table[LocalOffset] ^= inmsg;
        }
      } else if (status.MPI_TAG == FINISHED_TAG) {
        /* we got a done message.  Thanks for playing... */
        NumberReceiving--;
      } else {
        MPI_Abort( MPI_COMM_WORLD, -1 );
      }
      MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
                MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
    }
  } while (have_done && NumberReceiving > 0);

  MPI_Test(&outreq, &have_done, MPI_STATUS_IGNORE);
  if (have_done) {
    outreq = MPI_REQUEST_NULL;
    pe = HPCC_GetUpdates(Buckets, LocalSendBuffer, localBufferSize,
                         &peUpdates);
    MPI_Isend(&LocalSendBuffer, peUpdates, tparams.dtype64, (int)pe,
              UPDATE_TAG, MPI_COMM_WORLD, &outreq);
    pendingUpdates -= peUpdates;
  }
}
/* send our done messages */
for (proc_count = 0 ; proc_count < tparams.NumProcs ; ++proc_count) {
  if (proc_count == tparams.MyProc) { tparams.finish_req[tparams.MyProc] =
                                      MPI_REQUEST_NULL; continue; }
  /* send garbage - who cares, no one will look at it */
  MPI_Isend(&Ran, 0, tparams.dtype64, proc_count, FINISHED_TAG,
            MPI_COMM_WORLD, tparams.finish_req + proc_count);
}
/* Finish everyone else up... */
while (NumberReceiving > 0) {
  MPI_Wait(&inreq, &status);
  if (status.MPI_TAG == UPDATE_TAG) {
    MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
    bufferBase = 0;
    for (j=0; j < recvUpdates; j ++) {
      inmsg = LocalRecvBuffer[bufferBase+j];
      LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                     tparams.GlobalStartMyProc;
      HPCC_Table[LocalOffset] ^= inmsg;
    }
  } else if (status.MPI_TAG == FINISHED_TAG) {
    /* we got a done message.  Thanks for playing... */
    NumberReceiving--;
  } else {
    MPI_Abort( MPI_COMM_WORLD, -1 );
  }
  MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
            MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
}

MPI_Waitall( tparams.NumProcs, tparams.finish_req, tparams.finish_statuses);
```

HPCC RA: MPI kernel comment vs. Chapel

```
/* Perform updates to main table.  The scalar equivalent is:
 *
 *   for (i=0; i<NUPDATE; i++) {
 *     Ran = (Ran << 1) ^ (((s64Int) Ran < 0) ? POLY : 0);
 *     Table[Ran & (TABSIZE-1)] ^= Ran;
 *   }
 */
```

Chapel Kernel

```
forall (_, r) in zip(Updates, RAStream()) do
  T[r & indexMask].xor(r);
```

MPI Comment

```
/* Perform updates to main table. The scalar equivalent is:
 *
 *     for (i=0; i<NUPDATE; i++) {
 *       Ran = (Ran << 1) ^ (((s64Int) Ran < 0) ? POLY : 0);
 *       Table[Ran & (TABSIZE-1)] ^= Ran;
 *     }
 */
```

HPCC RA: Chapel translation

- Given the Chapel code:

```
forall (_, r) in zip(Updates, RAStream()) do
  T[r & indexMask].xor(r);
```

- An *approximate* translation of this code is:

```
coforall tid in 0..#nTasks do on … do                // create a number of distributed tasks
  for r in chunk(RAStream(), tid, nTasks) do         // loop over each task's iterations…
    T[r & indexMask].xor(r);                         // …computing each atomic op serially
```

HPCC RA: Chapel translation

- Given the Chapel code:

```
forall (_, r) in zip(Updates, RAStream()) do
  T[r & indexMask].xor(r);
```

- An *approximate* translation of this code is:

```
coforall tid in 0..#nTasks do on … do               // create a number of distributed tasks
  for r in chunk(RAStream(), tid, nTasks) do        // loop over each task's iterations…
    T[r & indexMask].xor(r);                        // …computing each atomic op serially
```

Note an opportunity for optimization:

- forall-loops imply iterations can execute simultaneously / in any order
- T[] is obviously not read again within this loop's body
- therefore, there's no need to serially execute each atomic op

HPCC RA: Chapel translation, optimized

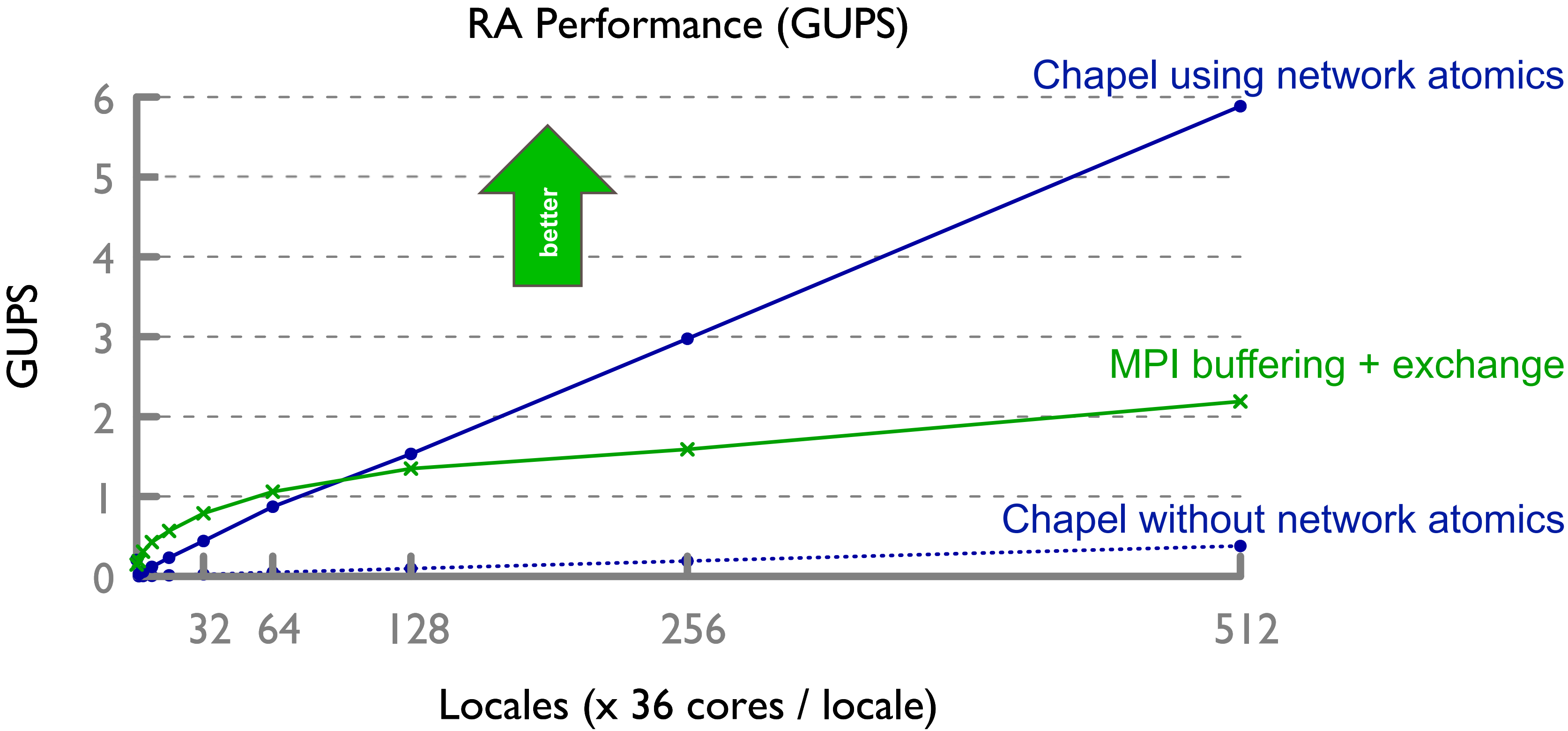
- Given the Chapel code:

```
forall (_, r) in zip(Updates, RAStream()) do
  T[r & indexMask].xor(r);
```

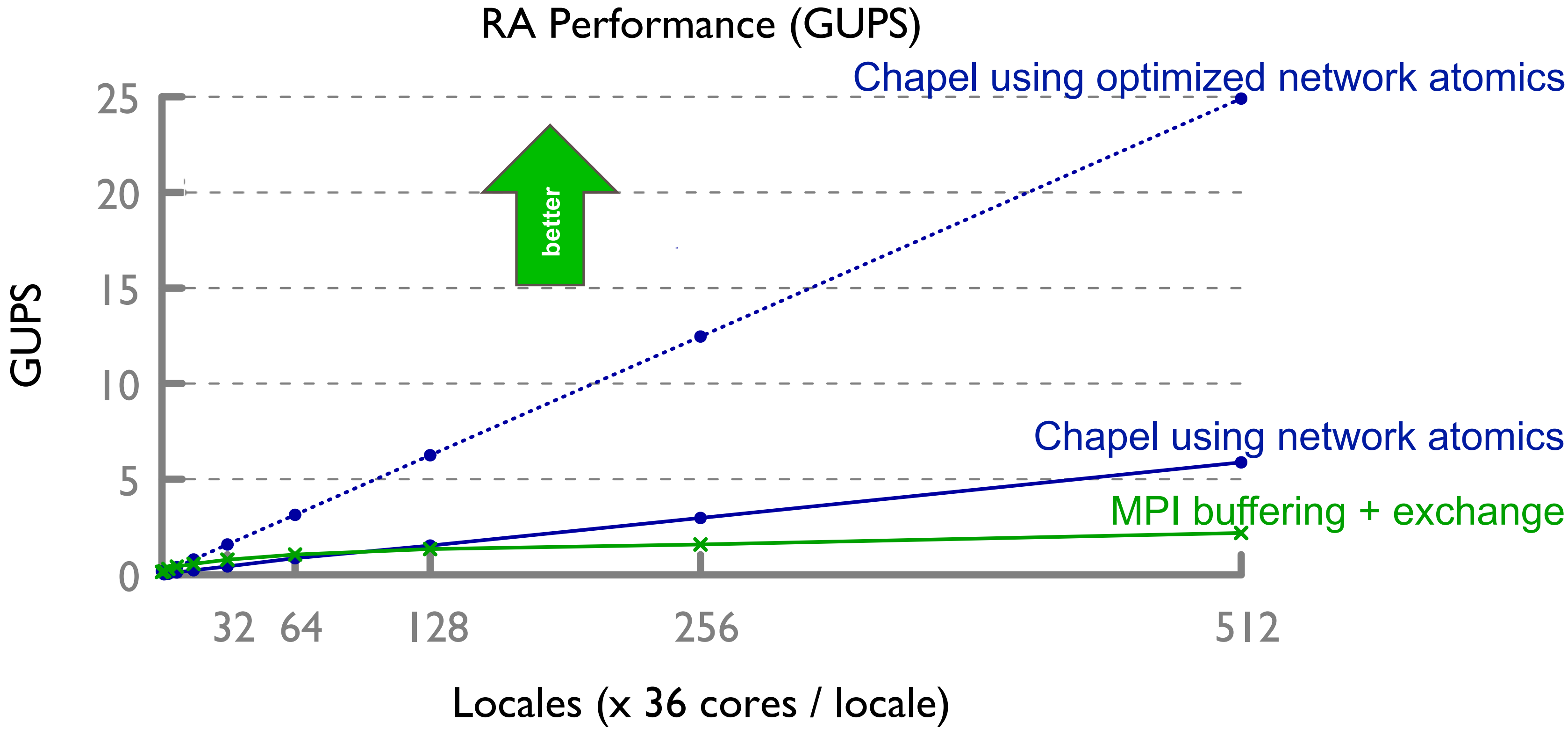
- An *approximate* translation of this code, when optimized, is:

```
coforall tid in 0..#nTasks do on … do            // create a number of distributed tasks
  for r in chunk(RAStream(), tid, nTasks) do     // loop over each task's iterations…
    T[r & indexMask].xor_async(r);               // …computing each atomic op asynchronously
  // tasks wait for asynchronous atomics to complete before terminating
```

HPCC RA: MPI vs. Chapel



HPCC RA: MPI vs. Chapel vs. Chapel optimized



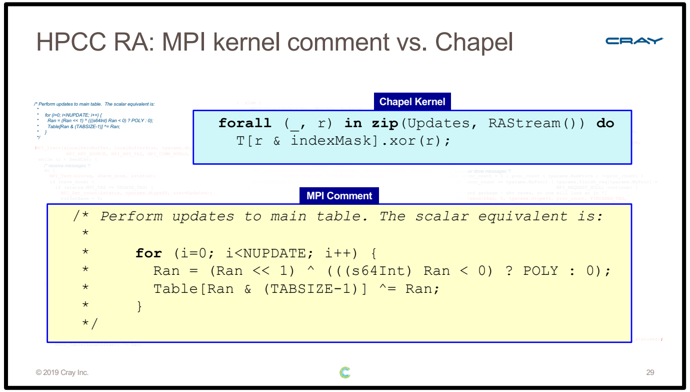
Notes on this optimization

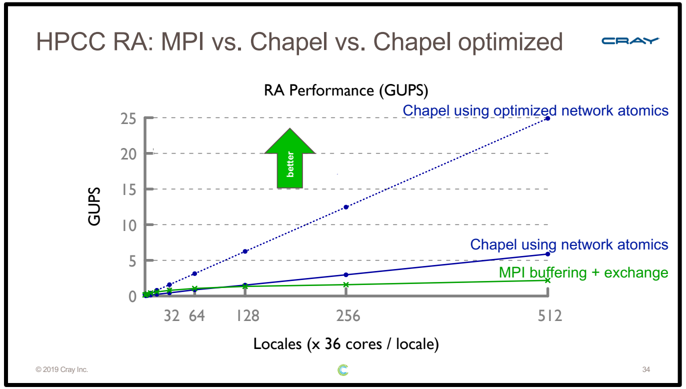
Of course, a human programmer could write our optimized version as well…

…but at what level of effort?

…and with what impact on performance portability?

Eventually, such comparisons become an arms race in which you have to decide where you stand in the “assembly vs. Fortran” style tradeoffs

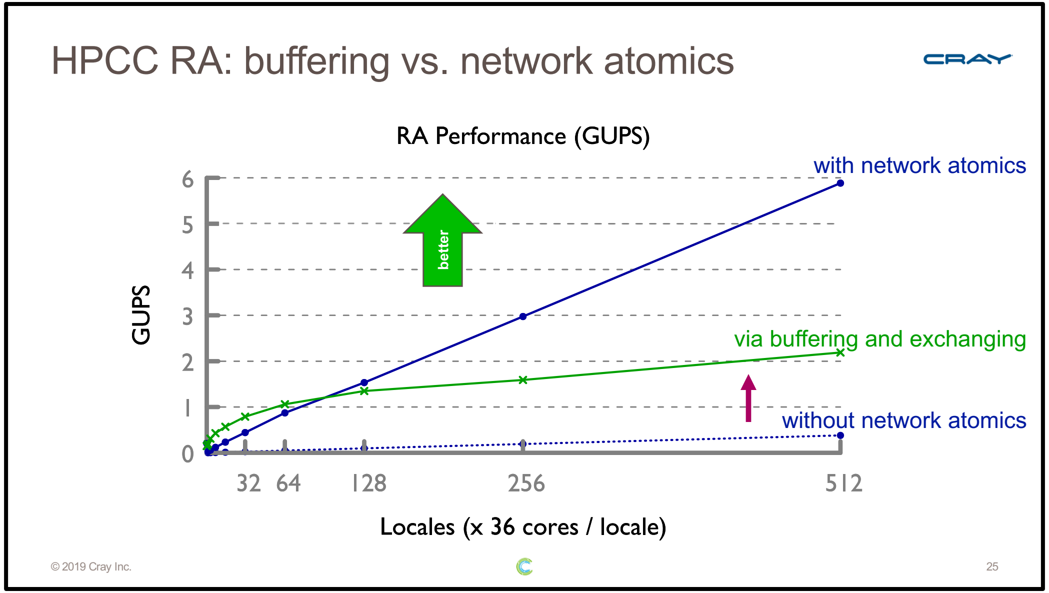




Notes on this optimization: Next Steps

Next Steps: similarly optimize no-network-atomics case

- **goal:** close gap with respect to performance of MPI version



Typical arguments against languages for HPC

- “It’s too difficult for new languages to get adopted”
- “We’re too small of a community to be able to support a language”
- “HPC programmers are happy with current programming methods”
- “HPC is so performance-oriented that productivity doesn’t matter”
- “It’s challenging to get performance from parallel languages”

I think there are counterarguments to each of these, the overarching one being:
“Scalable parallel programming is deserving of first-class language support”

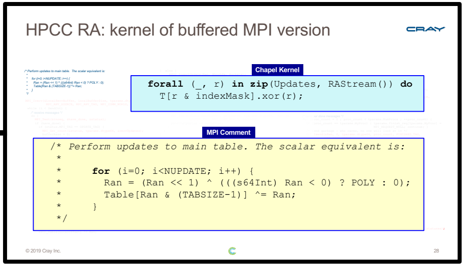
Why Consider New Languages at all?

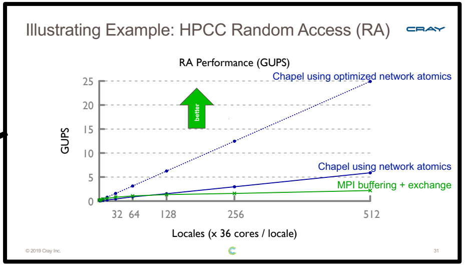
Syntax
- High level, elegant syntax
- Improve programmer productivity

Semantics
- Static analysis can help with correctness
- We need a compiler (front-end)

Performance
- If optimizations are needed to get performance
- We need a compiler (back-end)

Algorithms
- Language defines what is easy and hard
- Influences algorithmic thinking





[Source: Kathy Yelick, CHIUW 2018 keynote: *Why Languages Matter More Than Ever*]

Chapel's approach to performance portability

Language Design:

- Support direct expression of parallelism and locality
- Support abstraction of key high-level parallel idioms
  (e.g., parallel loops, distributed arrays)
- Support dropping to lower levels when necessary, including interoperation

Compiler Optimization:

- Map features to performance-oriented hardware features when available
  - make best effort translations when not
- Automatically optimize code based on semantics

Runtime Architecture:

- Runtime interfaces architected to support switching between implementations
  (e.g., communication over uGNI, ofi / libfabric, GASNet-EX)

What about numerical libraries?

- I haven't touched much on the "library" aspect of this minisymposium's theme
- My opinion is that parallel / distributed numerical libraries should be written in parallel / distributed languages, like Chapel
- In addition, Chapel has many features designed to help with engineering libraries
  - type inference / generic programming
  - object-orientation
  - rich procedure call support
  - managed memory
  - error-handling
  - …

The Chapel Team at Cray (May 2018)

~13 full-time employees + ~2 summer interns

Summary

True performance portability is challenging without giving up performance

Programming languages can significantly help with performance portability by raising the level of abstraction

- *simplifying coding and algorithmic exploration for users*
- *mapping to the best-available mechanisms on the target architecture*
- *enabling automatic optimizations*

HPC is overdue for its "assembly-to-Fortran" conversion moment

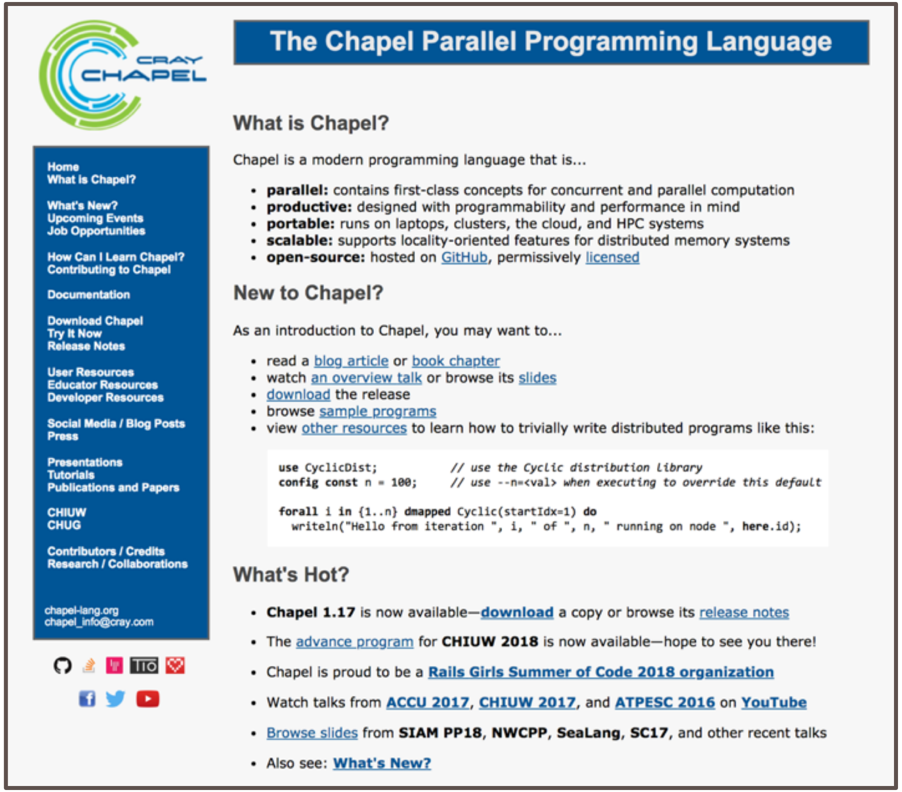
- *we believe Chapel is a key contender in support of such a switch*

Chapel Resources

Chapel Central

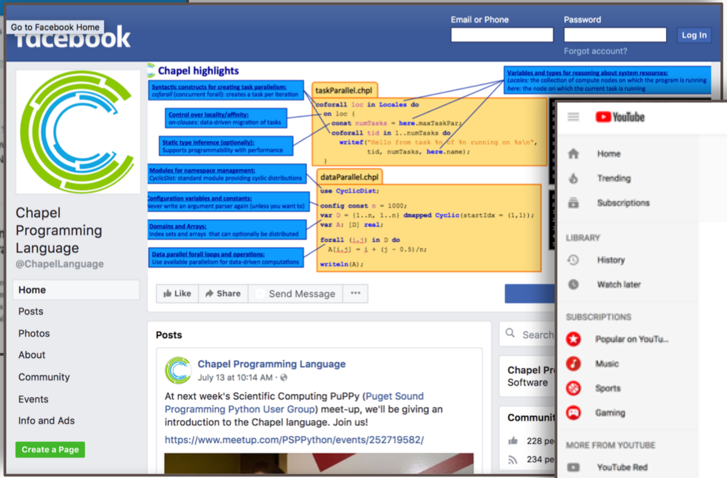
https://chapel-lang.org

- downloads
- presentations
- papers
- resources
- documentation

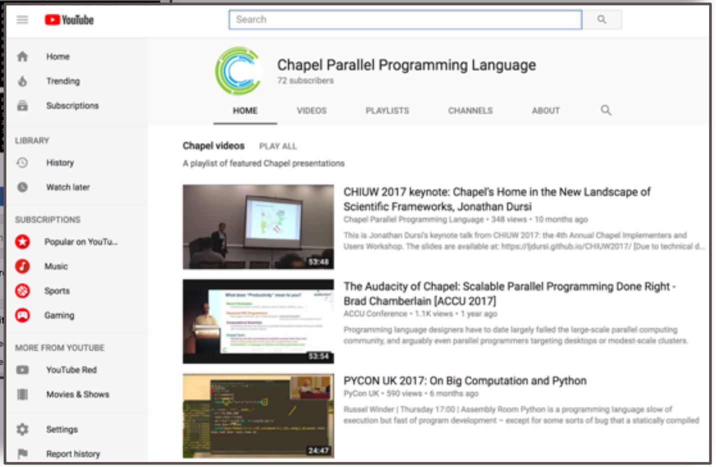


Chapel Social Media (no account required)

http://twitter.com/ChapelLanguage

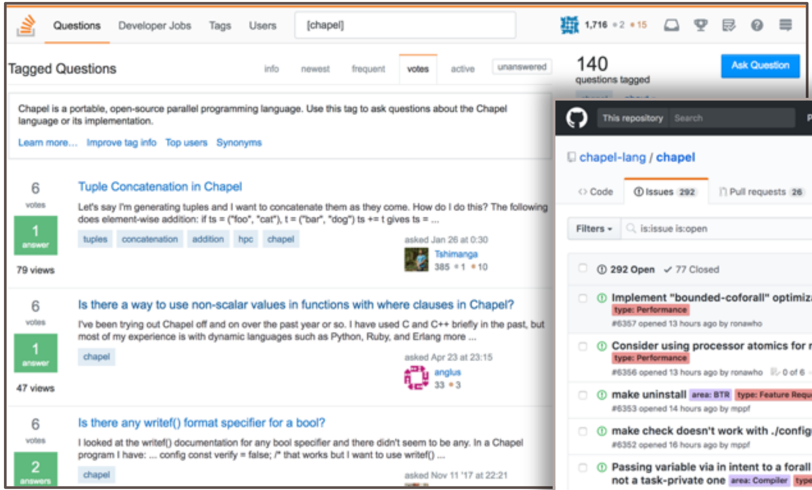


http://facebook.com/ChapelLanguage

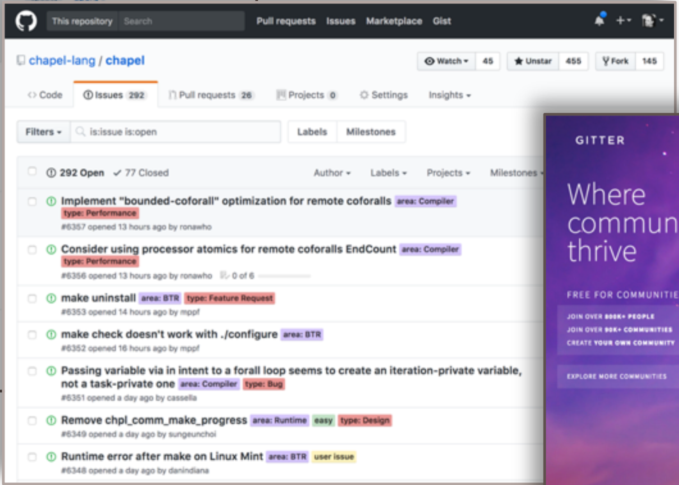


https://www.youtube.com/channel/UCHmm27bYjhknK5mU7ZzPGsQ/

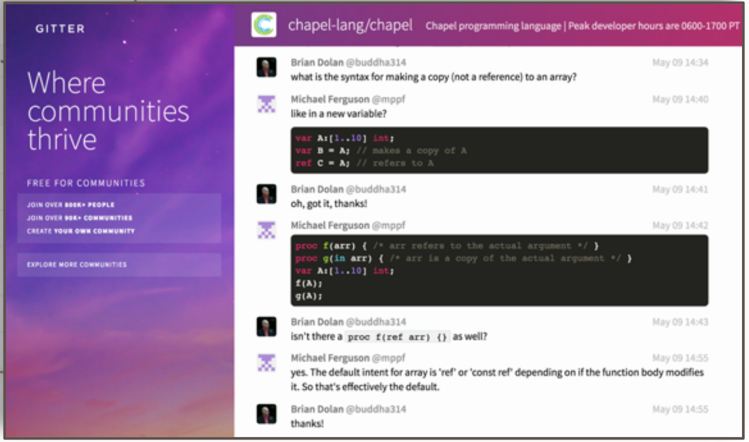
Chapel Community



https://stackoverflow.com/questions/tagged/chapel



https://github.com/chapel-lang/chapel/issues



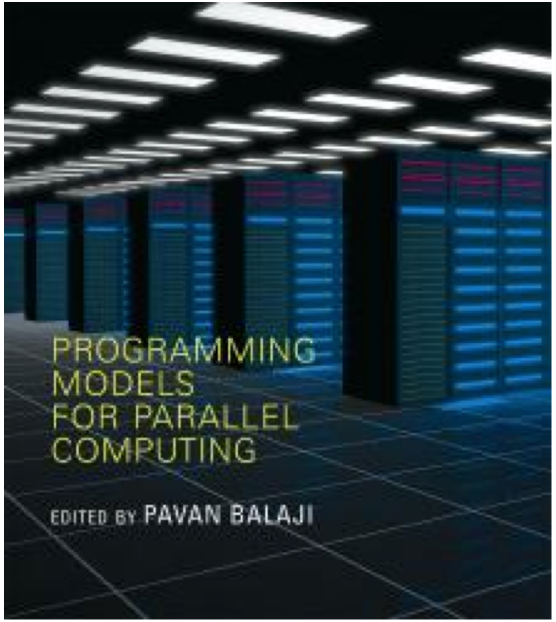
https://gitter.im/chapel-lang/chapel

read-only mailing list: chapel-announce@lists.sourceforge.net (~15 mails / year)

Suggested Reading: Chapel history and overview

Chapel chapter from *Programming Models for Parallel Computing*

- a detailed overview of Chapel's history, motivating themes, features
- published by MIT Press, November 2015
- edited by Pavan Balaji (Argonne)
- chapter is also available online



Suggested Reading: Recent Progress (CUG 2018)

Chapel Comes of Age: Making Scalable Programming Productive

Bradford L. Chamberlain, Elliot Ronaghan, Ben Albrecht, Lydia Duncan, Michael Ferguson,
Ben Harshbarger, David Iten, David Keaton, Vassily Litvinov, Preston Sahabu, and Greg Titus
Chapel Team
Cray Inc.
Seattle, WA, USA
chapel_info@cray.com

***Abstract*—Chapel is a programming language whose goal is to support productive, general-purpose parallel computing at scale. Chapel's approach can be thought of as combining the strengths of Python, Fortran, C/C++, and MPI in a single language. Five years ago, the DARPA High Productivity Computing Systems (HPCS) program that launched Chapel wrapped up, and the team embarked on a five-year effort to improve Chapel's appeal to end-users. This paper follows up on our CUG 2013 paper by summarizing the progress made by the Chapel project since that time. Specifically, Chapel's performance now competes with or beats hand-coded C+MPI/SHMEM+OpenMP; its suite of standard libraries has grown to include FFTW, BLAS, LAPACK, MPI, ZMQ, and other key technologies; its documentation has been modernized and fleshed out; and the set of tools available to Chapel users has grown. This paper also characterizes the experiences of early adopters from communities as diverse as astrophysics and artificial intelligence.**

***Keywords*-Parallel programming; Computer languages**

I. Introduction

Chapel is a programming language designed to support productive, general-purpose parallel computing at scale. Chapel's approach can be thought of as striving to create a language whose code is as attractive to read and write as Python, yet which supports the performance of Fortran and the scalability of MPI. Chapel also aims to compete with C in terms of portability, and with C++ in terms of flexibility and extensibility. Chapel is designed to be general-purpose in the sense that when you have a parallel algorithm in mind and a parallel system on which you wish to run it, Chapel should be able to handle that scenario.

Chapel's design and implementation are led by Cray Inc. with feedback and code contributed by users and the open-source community. Though developed by Cray, Chapel's design and implementation are portable, permitting its programs to scale up from multicore laptops to commodity clusters to Cray systems. In addition, Chapel programs can be run on cloud-computing platforms and HPC systems from other vendors. Chapel is being developed in an open-source manner under the Apache 2.0 license and is hosted at GitHub.[1]

The development of the Chapel language was undertaken by Cray Inc. as part of its participation in the DARPA High Productivity Computing Systems program (HPCS). HPCS wrapped up in late 2012, at which point Chapel was a compelling prototype, having successfully demonstrated several key research challenges that the project had undertaken. Chief among these was supporting data- and task-parallelism in a unified manner within a single language. This was accomplished by supporting the creation of high-level data-parallel abstractions like parallel loops and arrays in terms of lower-level Chapel features such as classes, iterators, and tasks.

Under HPCS, Chapel also successfully supported the expression of parallelism using distinct language features from those used to control locality and affinity—that is, Chapel programmers specify *which* computations should run in parallel distinctly from specifying *where* those computations should be run. This permits Chapel programs to support multicore, multi-node, and heterogeneous computing within a single unified language.

Chapel's implementation under HPCS demonstrated that the language could be implemented portably while still being optimized for HPC-specific features such as the RDMA support available in Cray® Gemini™ and Aries™ networks. This allows Chapel to take advantage of native hardware support for remote puts, gets, and atomic memory operations.

Despite these successes, at the close of HPCS, Chapel was not at all ready to support production codes in the field. This was not surprising given the language's aggressive design and modest-sized research team. However, reactions from potential users were sufficiently positive that, in early 2013, Cray embarked on a follow-up effort to improve Chapel and move it towards being a production-ready language. Colloquially, we refer to this effort as "the five-year push."

This paper's contribution is to describe the results of this five-year effort, providing readers with an understanding of Chapel's progress and achievements since the end of the HPCS program. In doing so, we directly compare the status of Chapel version 1.17, released last month, with Chapel version 1.7, which was released five years ago in April 2013.

[1] https://github.com/chapel-lang/chapel

paper and slides available at chapel-lang.org

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.

THANK YOU

QUESTIONS?

bradc@cray.com
@ChapelLanguage
chapel-lang.org

cray.com
@cray_inc
linkedin.com/company/cray-inc-