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?
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
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
parallel for val in RandomValues:
    loc ← val & mask
    Table[loc] ← Table[loc] atomic-xor val
parallel for val in RandomValues:
  loc ← val & mask
  Table[loc] ← Table[loc] atomic-xor val
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 atoms
HPCC RA: with or without network atomics

RA Performance (GUPS)

Locales (x 36 cores / locale)

GUPS

with network atomics

without network atomics

better

© 2019 Cray Inc.
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

RA Performance (GUPS)

Locales (x 36 cores / locale)

with network atomics
via buffering and exchanging
without network atomics
better

© 2019 Cray Inc.
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

RA Performance (GUPS)

Locales (x 36 cores / locale)

with network atomics
via buffering and exchanging
without network atomics

better
HPCC RA: MPI vs. Chapel

RA Performance (GUPS)

Chapel using network atomics
Chapel without network atomics
MPI buffering + exchange

Chapel with network atomics
Chapel without network atomics

Locales (x 36 cores / locale)

GUPS
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.
/* Perform updates to main table. The scalar equivalent is:

* for (i0 = UPDATE; i0 < MAXUPDATE; i0++)
  * Ran = (Ran << 1) ^ (((s64Int) Ran < 0) ? POLY : 0);
  */

while (i < SendCnt) {
  if (do {
    for (i=0; i<NUPDATE; i++) {
      Table[Ran & (TABSIZE - 1)] ^= Ran;
    } else
      if (MPI_Test(&inreq, &have_done)
        proc_count > 0) {
        for (j=0; j < recvUpdates; j++) {
          inmsg = LocalRecvBuffer[bufferBase+j];
          if (inmsg & (tparams.TableSize - 1))
            do {
              proc_count = 0;
              proc_count -= 1;
            } else {
              MPI_Abort (MPI_COMM_WORLD, -1);
            } MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
          pendingUpdates += peUpdates;
        } else if (status.MPI_TAG == FINISHED_TAG)
          if (have_done) {
            proc_count = 0;
            proc_count -= 1;
          } else {
            MPI_Abort (MPI_COMM_WORLD, -1);
          } MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
          pendingUpdates += peUpdates;
        } else if (status.MPI_TAG == UPDATE_TAG)
          do {
            pendingUpdates += maxPendingUpdates;
          } else {
            HPCC_InsertUpdate(Ran, WhichPw, BCuels); pendingUpdates++; }
        i++;
      } else {
        HPCC_InsertUpdate(Ran, WhichPw, Buckets); pendingUpdates++; }
    }
  }

MPI_Test(&inreq, have_done, &status); if (have_done) {
  HPCC_InsertUpdate(Ran, WhichPw, Buckets);
  pendingUpdates++; }
  i++;
}

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
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(4Ran, 0, tparams.dtype64, proc_count, FINISHED_TAG, MPI_COMM_WORLD, tparams.finish_req + proc_count);
}

/* Finish everyone else up... */
while (NumberReceiving > 0) {
  if (status.MPI_TAG == UPDATE_TAG)
    do {
      bufferBase = 0;
      for (j=0; j < recvUpdates; j++) {
        inmsg = LocalRecvBuffer[bufferBase+j];
        offset = (inmsg & (tparams.TableSize - 1)) - tparams.GlobalStartMyProc;
        HPCC_Table[localOffset] = inmsg;
      }
    } else if (status.MPI_TAG == FINISHED_TAG)
      if (have_done) {
        bufferBase = 0;
        for (j=0; j < recvUpdates; j++)
          do {
            NumberReceiving -= 1;
          } else {
            proc_count = 0;
            proc_count -= 1;
          } MPI_Abort (MPI_COMM_WORLD, -1);
        MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
        pendingUpdates += peUpdates;
      } else {
        proc_count = 0;
        proc_count -= 1;
      } MPI_Abort (MPI_COMM_WORLD, -1);
    } MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
    pendingUpdates += peUpdates;
}

MPI_Test(&inreq, have_done, &status); if (have_done) {
  outreq = MPI_REQUEST_NULL;
  pw = HPCC_GetUpdates(Buckets, LocalSendBuffer, localBufferSize, &peUpdates);
  MPI_Isend(LocalSendBuffer, peUpdates, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
  pendingUpdates += peUpdates;
}

/* we got a done message. Thanks for playing... */
else {
  proc_count = 0;
  proc_count -= 1;
}

MPI_Abort (MPI_COMM_WORLD, -1);

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
pendingUpdates += peUpdates;

/* we got a done message. Thanks for playing... */
else {
  proc_count = 0;
  proc_count -= 1;
}

MPI_Abort (MPI_COMM_WORLD, -1);

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
pendingUpdates += peUpdates;

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
pendingUpdates += peUpdates;

MPI_Abort (MPI_COMM_WORLD, -1);

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
pendingUpdates += peUpdates;

MPI_Abort (MPI_COMM_WORLD, -1);

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
pendingUpdates += peUpdates;

MPI_Abort (MPI_COMM_WORLD, -1);

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
pendingUpdates += peUpdates;

MPI_Abort (MPI_COMM_WORLD, -1);

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
pendingUpdates += peUpdates;

MPI_Abort (MPI_COMM_WORLD, -1);

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
pendingUpdates += peUpdates;

MPI_Abort (MPI_COMM_WORLD, -1);

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
pendingUpdates += peUpdates;

MPI_Abort (MPI_COMM_WORLD, -1);

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
pendingUpdates += peUpdates;

MPI_Abort (MPI_COMM_WORLD, -1);

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
pendingUpdates += peUpdates;

MPI_Abort (MPI_COMM_WORLD, -1);

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
pendingUpdates += peUpdates;

MPI_Abort (MPI_COMM_WORLD, -1);

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
pendingUpdates += peUpdates;

MPI_Abort (MPI_COMM_WORLD, -1);

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
pendingUpdates += peUpdates;

MPI_Abort (MPI_COMM_WORLD, -1);

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
pendingUpdates += peUpdates;

MPI_Abort (MPI_COMM_WORLD, -1);

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
pendingUpdates += peUpdates;

MPI_Abort (MPI_COMM_WORLD, -1);

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
pendingUpdates += peUpdates;

MPI_Abort (MPI_COMM_WORLD, -1);

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
pendingUpdates += peUpdates;

MPI_Abort (MPI_COMM_WORLD, -1);

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pi, UPDATE_TAG, MPI_COMM_WORLD, &inreq);
pendingUpdates += peUpdates;

MPI_Abort (MPI_COMM_WORLD, -1);
/* 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;
}

forall (_, r) in zip(Updates, RAStream()) do
    T[r & indexMask].xor(r);
HPCC RA: Chapel translation

• Given the Chapel code:

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

• An approximate translation of this code is:

```chapel
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:

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

• An approximate translation of this code is:

```chapel
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:

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

• An approximate translation of this code, when optimized, is:

```chapel
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

RA Performance (GUPS)

GUPS

Locales (x 36 cores / locale)

Chapel using network atomics

MPI buffering + exchange

Chapel without network atomics

better
HPCC RA: MPI vs. Chapel vs. Chapel optimized

RA Performance (GUPS)

Chapel using optimized network atomics
Chapel using network atomics
MPI buffering + exchange

Locales (x 36 cores / locale)

GUPS
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, CHI/UW 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)

- [Twitter](http://twitter.com/ChapelLanguage)
- [Facebook](http://facebook.com/ChapelLanguage)
- [YouTube](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
Chapel Comes of Age: Making Scalable Programming Productive

Bradford L. Chamberlain, Elliot Ronayagh, Ben Albrecht, Lydia Duncan, Michael Ferguson, Ben Harmsberger, David Ier, David Kohut, Vasily Litvinov, Prasoon Sabnani, and Greg Titze
Chapel Team
Cray Inc.
Seattle, WA, USA
chapel-info@cray.com

Abstract—Chapel is a programming language whose goal is to support productive parallel computing at scale. Chapel’s approach can be thought of as combining the strengths of Fortran, Python, C++, and MPI in a single language. Five years ago, the DARPA High Productivity Computing Systems (HPCS) program that launched Chapel aimed to create a new language that would simplify writing parallel programs and improve programmer productivity. This paper documents the progress made by the Chapel project since that time. Specifically, Chapel’s performance on complex workloads with hardware vectorized operations has improved significantly using both Chapel’s built-in high-level vectorization and traditional loop-based vectorization. The language, compiler and runtime have also been extended to support new features such as out-of-core and hybrid parallelism. Chapel supports data parallelism with a native map-reduce construct and is also available as a web-based environment through the Chapel Cloud. This paper documents the improvements made over the past five years to improve Chapel’s support for out-of-core computing, Chapel’s ability to support vector operations and Chapel’s ability to support both async and sync modes of execution. Chapel code is available in source code form on GitHub.

1. INTRODUCTION
Chapel is a programming language designed to support productive, general-purpose parallel computing at scale. Chapel’s approach can be thought of as combining the strengths of Fortran, Python, C++, and MPI in a single language. Five years ago, the DARPA High Productivity Computing Systems (HPCS) program that launched Chapel aimed to create a new language that would simplify writing parallel programs and improve programmer productivity. This paper documents the progress made by the Chapel project since that time. Specifically, Chapel’s performance on complex workloads with hardware vectorized operations has improved significantly using both Chapel’s built-in high-level vectorization and traditional loop-based vectorization. The language, compiler and runtime have also been extended to support new features such as out-of-core and hybrid parallelism. Chapel supports data parallelism with a native map-reduce construct and is also available as a web-based environment through the Chapel Cloud. This paper documents the improvements made over the past five years to improve Chapel’s support for out-of-core computing, Chapel’s ability to support vector operations and Chapel’s ability to support both async and sync modes of execution. Chapel code is available in source code form on GitHub.

2. BACKGROUND
Chapel’s implementation uses the HPCs program to implement parallelism while still being optimized for HPC-specific features such as RDMA support available in Cray® Gemini™ and Adirondack™ systems. This allows Chapel to take advantage of native hardware support for remote pairs, gpus, and atomic memory operations.

Despite these successes, the goal of Chapel was not to be an all-in-one solution for productive parallel computing in the field. This was not surprising given the language’s expressive design and the desire to keep the core Chapel language and runtime simple. As a result, Chapel was designed to be a feature-rich compiler that can be used to implement both Chapel and other languages in parallel. Chapel also supports vector operations with a native map-reduce construct and is also available as a web-based environment through the Chapel Cloud. This paper documents the improvements made over the past five years to improve Chapel’s support for out-of-core computing, Chapel’s ability to support vector operations and Chapel’s ability to support both async and sync modes of execution. Chapel code is available in source code form on GitHub.

3. RESULTS
Chapel’s implementation used the HPCs program to implement parallelism while still being optimized for HPC-specific features such as RDMA support available in Cray® Gemini™ and Adirondack™ systems. This allows Chapel to take advantage of native hardware support for remote pairs, gpus, and atomic memory operations.

Despite these successes, the goal of Chapel was not to be an all-in-one solution for productive parallel computing in the field. This was not surprising given the language’s expressive design and the desire to keep the core Chapel language and runtime simple. As a result, Chapel was designed to be a feature-rich compiler that can be used to implement both Chapel and other languages in parallel. Chapel also supports vector operations with a native map-reduce construct and is also available as a web-based environment through the Chapel Cloud. This paper documents the improvements made over the past five years to improve Chapel’s support for out-of-core computing, Chapel’s ability to support vector operations and Chapel’s ability to support both async and sync modes of execution. Chapel code is available in source code form on GitHub.

4. FUTURE WORK
Chapel’s implementation used the HPCs program to implement parallelism while still being optimized for HPC-specific features such as RDMA support available in Cray® Gemini™ and Adirondack™ systems. This allows Chapel to take advantage of native hardware support for remote pairs, gpus, and atomic memory operations.

Despite these successes, the goal of Chapel was not to be an all-in-one solution for productive parallel computing in the field. This was not surprising given the language’s expressive design and the desire to keep the core Chapel language and runtime simple. As a result, Chapel was designed to be a feature-rich compiler that can be used to implement both Chapel and other languages in parallel. Chapel also supports vector operations with a native map-reduce construct and is also available as a web-based environment through the Chapel Cloud. This paper documents the improvements made over the past five years to improve Chapel’s support for out-of-core computing, Chapel’s ability to support vector operations and Chapel’s ability to support both async and sync modes of execution. Chapel code is available in source code form on GitHub.
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.