The Future of PGAS Programming from a Chapel Perspective

Brad Chamberlain

PGAS 2015, Washington DC

September 17, 2015

(or:)
Five Things You Should Do to Create a Future-Proof Exascale Language

Brad Chamberlain

PGAS 2015, Washington DC

September 17, 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.

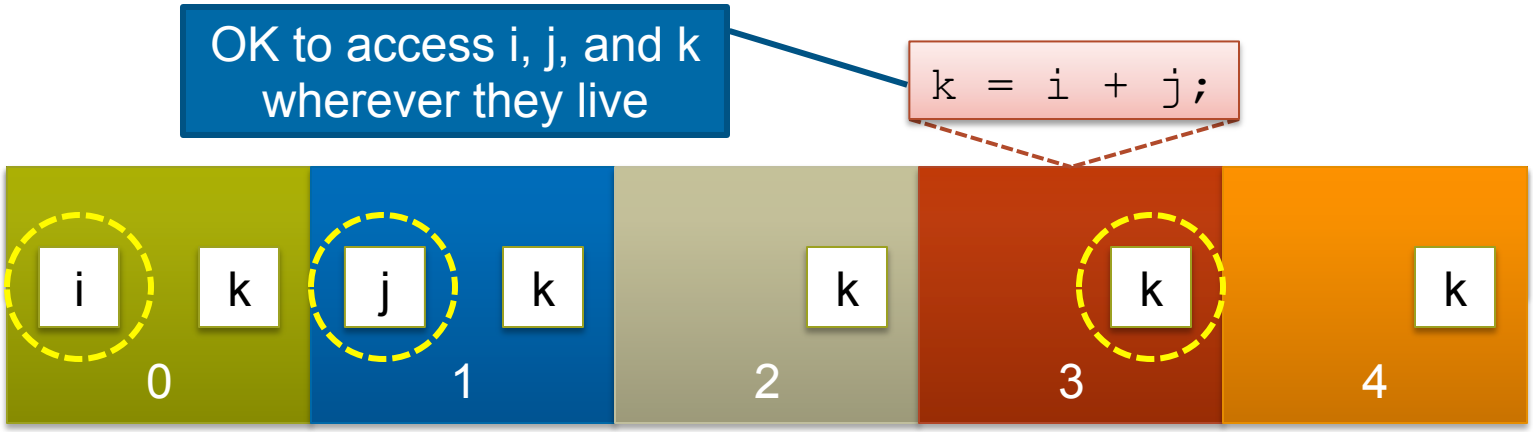
Apologies in advance

- This is a (mostly) brand-new talk
- My new talks tend to be overly text-heavy (sorry)

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”)

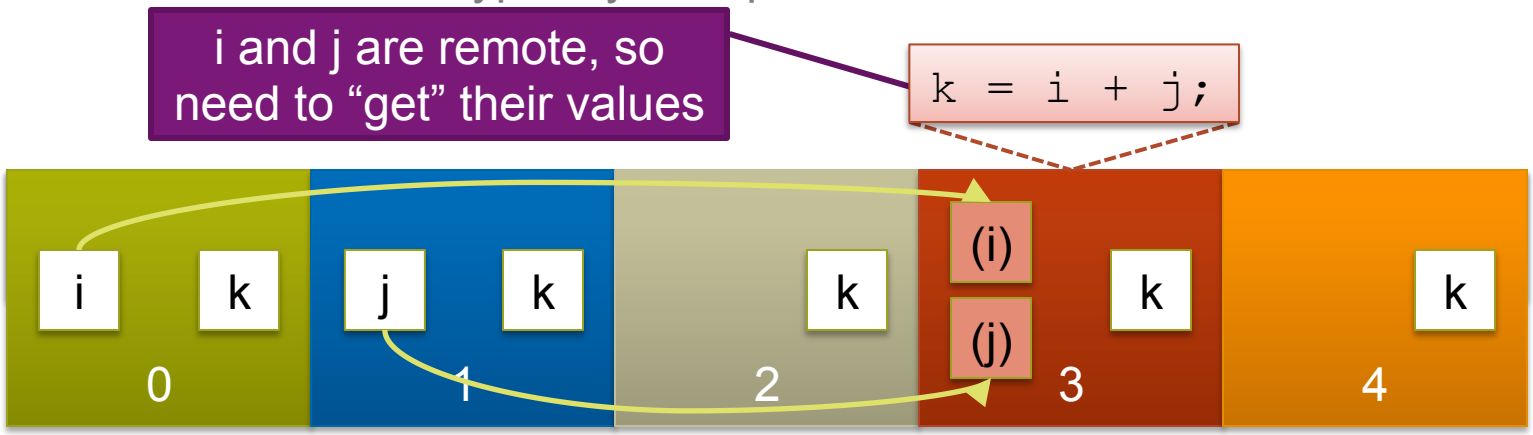
PGAS Programming in a Nutshell

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")

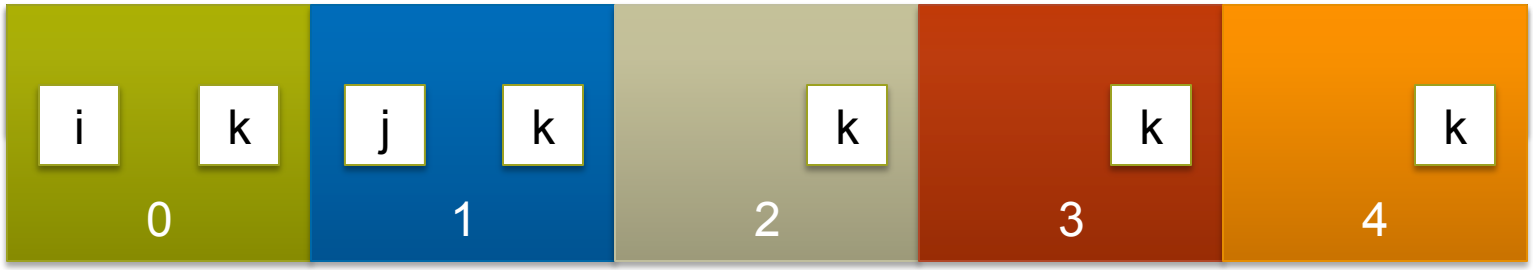
PGAS Programming in a Nutshell

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:

- 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



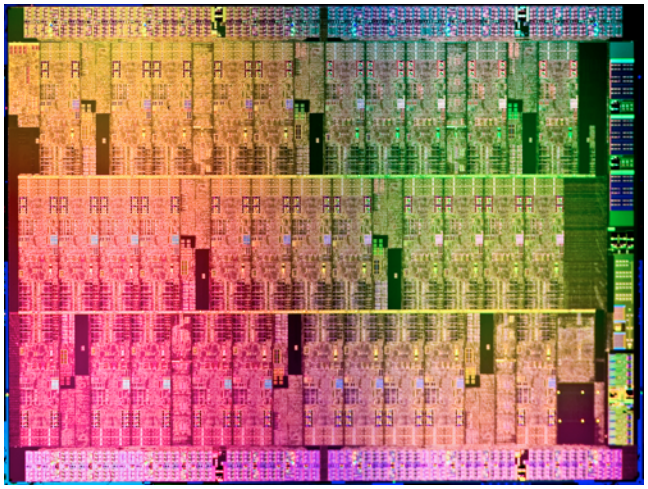
Images / Threads / Locales / Places / etc. (think: “compute nodes”)

A Brief History of PGAS Programming

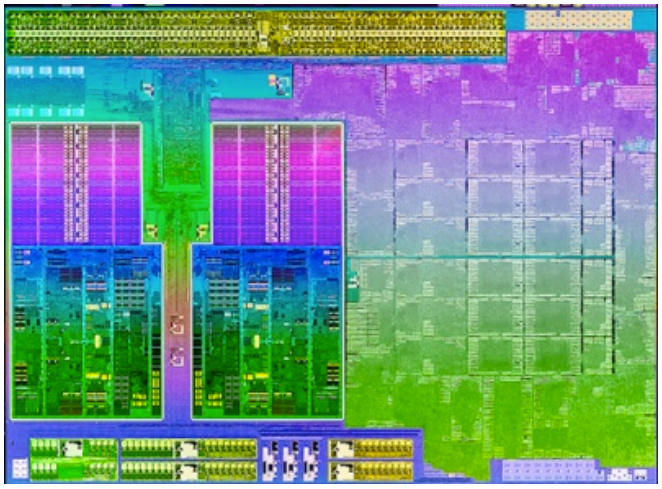
- **Founding Members:**
  - Co-Array Fortran, Unified Parallel C, and Titanium
    - SPMD dialects of Fortran, C, and Java, respectively
    - though quite different, shared a common characteristic: PGAS
    - coined term and banded together for “strength in numbers” benefits
- **Community response:**
  - Some intrigue and uptake, but generally not much broad user adoption
    - IMO, not enough benefit, flexibility, and value-add to warrant leaving MPI
- **Since then:**
  - Additional models have self-identified as PGAS
    - some retroactively: Global Arrays, ZPL, HPF, …
    - other new ones: Chapel, X10, XcalableMP, UPC++, Coarray C++, HPX, …
  - PGAS conference established (started 2005)
  - PGAS booth and BoF annually at SCxy (started ~SC07)
  - PGAS is taken more seriously, yet hasn’t hit a tipping point

(and by now, probably dated)

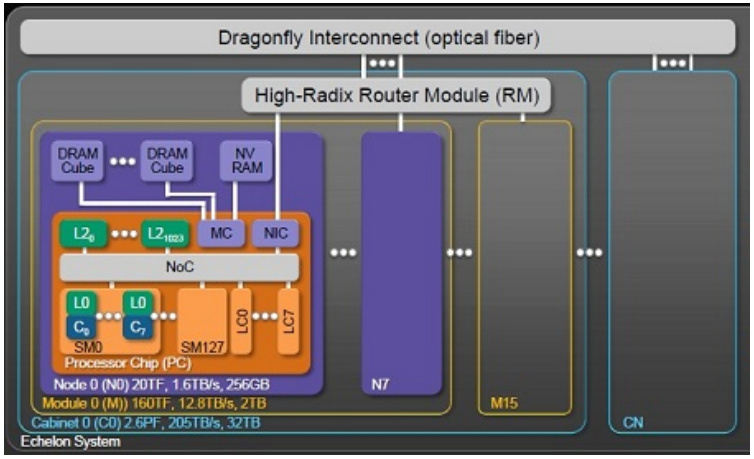
The Obligatory "Exascale is Coming!" Slide



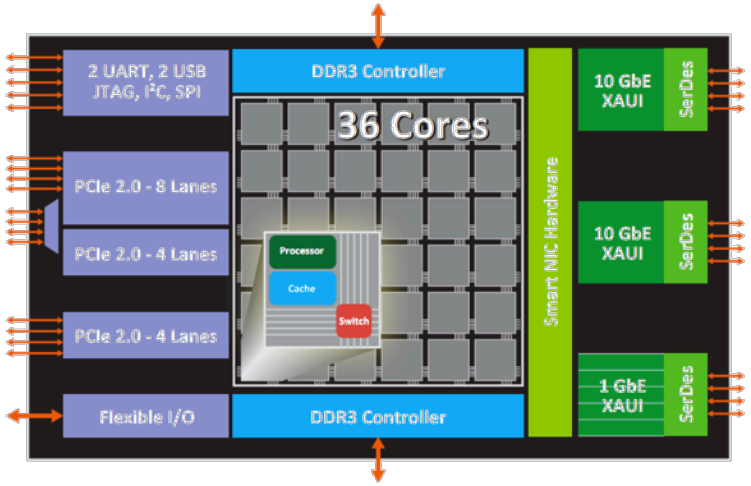
Intel Xeon Phi



AMD APU



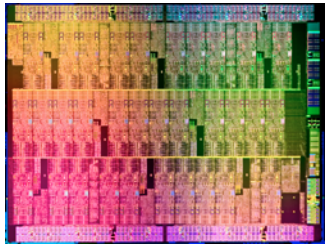
Nvidia Echelon

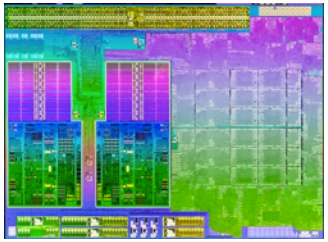


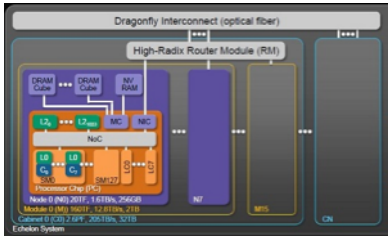
Tilera Tile-Gx

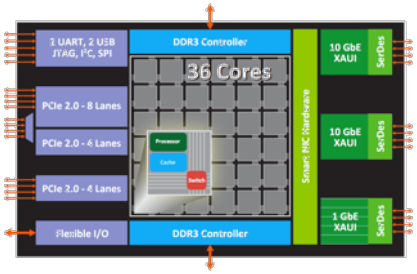
http://download.intel.com/pressroom/images/Aubrey_Isle_die.jpg http://www.zdnet.com/amds-trinity-processors-take-on-intels-ivy-bridge-3040155225/
http://insidehpc.com/2010/11/26/nvidia-reveals-details-of-echelon-gpu-designs-for-exascale/ http://tilera.com/sites/default/files/productbriefs/Tile-Gx%203036%20SB012-01.pdf

General Characteristics of These Architectures









- **Increased hierarchy and/or sensitivity to locality**
- **Potentially heterogeneous processor/memory types**

⇒ Next-gen programmers will have a lot more to think about at the node level than in the past

Concern over these challenges has generated [re]new[ed] interest in PGAS programming from some circles

So:
What do we need to do to create a successful PGAS language for exascale?
(or even: at all?)

But first, what are [y]our goals?

- **Create a successful PGAS language!**

- **Well yes, but to what end?**
  - To work on something because it's fun / a good challenge?
  - To publish papers?
  - To advance to the next level (degree, job, rank, …)?
  - To influence an existing language's standard?
  - To create something that will be practically adopted?
  - To create something that will revolutionize parallel programming?

My personal focus/bias is here.

If the others also happen along the way, bonus!

Five Things You Should Do to Create a Successful (exascale) PGAS Language

#1: Move Beyond SPMD Programming

SPMD: the status quo

- **SPMD parallelism is seriously weak sauce**
  - Think about parallel programming as you learned it in school:
    - divide-and-conquer
    - producer-consumer
    - streaming / pipeline parallelism
    - data parallelism
  - Contrast this with SPMD:
    - "You have 'p' parallel tasks, each of which runs for the program's duration."
    - "Period."

- **Don't lean on hybrid parallelism as a crutch**
  - Division of labor is a reasonable tactic…

    …but ultimately, hybrid models should be an option that's available, not a requirement for using parallel hardware effectively

Hybrid Models: Why?

Q: Why do we rely on hybrid models so much?

A: Our programming models have typically only handled…

…specific types of hardware parallelism

…specific units of software parallelism

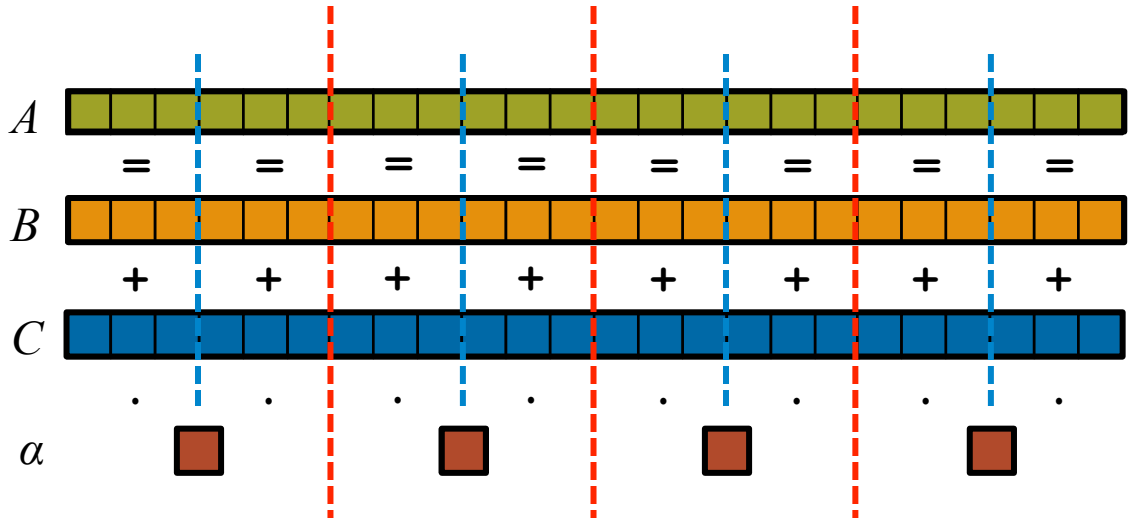
Type of HW Parallelism	Programming Model	Unit of Parallelism
Inter-node	MPI / traditional PGAS	executable
Intra-node/multicore	OpenMP / Pthreads	iteration/task
Instruction-level vectors/threads	pragmas	iteration
GPU/accelerator	Open[MP|CL|ACC] / CUDA	SIMD function/task

STREAM Triad: a trivial parallel computation

Given: m-element vectors A, B, C

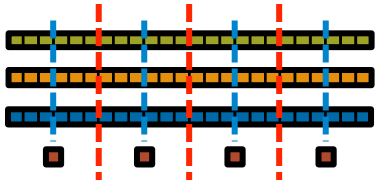
Compute: $\forall i \in 1..m,\ A_i = B_i + \alpha \cdot C_i$

Pictorially (hybrid distributed/shared memory version):



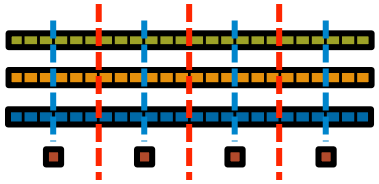
STREAM Triad: MPI vs. OpenMP vs. CUDA

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] = 0.0;
  }

  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 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;

  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];
}
```

As we look at more interesting computations, the differences only increase.

HPC suffers from too many distinct notations for expressing parallelism, due to designing programming models in an architecture-up way.

By Analogy: Let's Cross the United States!

By Analogy: Let's Cross the United States!

…Hey, what's that sound?

SPMD and Exascale

- **As weak as SPMD is today, exascale makes it worse**
  - Do you really want a process per core as core counts climb?
    - heavyweight
    - exacerbates surface-to-volume concerns in algorithms
      - and memory's not becoming any faster…
    - treats increasingly hierarchical machine as ridiculously flat

#1: Post-SPMD Challenge

Your Challenge: Create PGAS languages that are general, supporting unified concepts for expressing parallelism across diverse hardware types and software granularities.

Type of HW Parallelism	Programming Model	Unit of Parallelism
Inter-node	Your PGAS model	executable
Intra-node/multicore	Your PGAS model	iteration/task
Instruction-level vectors/threads	Your PGAS model	iteration
GPU/accelerator	Your PGAS model	SIMD function/task

#2: Support Locality Distinctly from Parallelism

Locality vs. Parallelism: Defining our Terms

The two primary concerns of scalable parallel programmers:

Parallelism: What should execute simultaneously?

Locality: Where should it run?

- **Do you agree that these are two distinct concerns?**
  - So why do we so rarely treat them as such?

Locality vs. Parallelism: the status quo

- **SPMD models conflate these concerns:**
  - the SPMD image is both the unit of both parallelism and locality
  - again, mixing in distinct programming models ≠ “success”
- **In other models, locality is essentially ignored**
  - e.g., OpenMP (traditionally), Pthreads
- **CUDA, Open[CL | ACC | MP 4.0] do better than most**
  - yet, these are niche—not intended for general use

Locality vs. Parallelism at Exascale

- **Increased hierarchy/heterogeneity will generate increased desire to address these concerns separately**
  - "This should execute in parallel"
  - "This should be allocated in that memory"
  - "This should be forked off to execute on that processor"

#2: Locality vs. Parallelism Challenge

Your Challenge: Create PGAS languages that let programmers express parallelism and locality concerns distinctly from one another.

- **We're PGAS, dealing with locality is our bread and butter**
  - we have no excuse not to be leading the charge here

#3: Get Communication-Syntax Relationship Right

Communication and Syntax: the status quo

PGAS languages traditionally fall into one of two camps:

1. syntactically visible communication:
   - e.g., Fortran's Co-Arrays:

     ```
     k = i[0] + j[1]
     ```
   - e.g., ZPL's operators on distributed arrays:

     ```
     [1..n-1] A = A@east;
     ```

   benefits:
   - + fewer surprises due to unexpected communication
   - + can help programmer reason about "cost" of their algorithm
2. syntax-free communication:
   - e.g., UPC's shared arrays and variables:

     ```
     k = i + j;
     ```
   - e.g., HPF's distributed arrays

     ```
     A[1:n-1] = A[2:n]
     ```

   benefits:
   - + arguably simpler, more natural
   - + easier to re-use code between shared- and distributed-memory executions

Visible vs. Invisible Communication

To date, the PGAS community has been very tolerant and accepting of both approaches.

No longer. I'm taking a stand!

Syntax-free communication is the way to go.

#3-redux: Support Communication Without Syntax

Support Syntax-Free Communication

Why?

- The “code re-use” argument trumps the others
  - Requiring users to rewrite their algorithms each time a data structure changes from local to distributed is for chumps / MPI programmers
- There are other good techniques for reasoning about locality
  - semantics / execution models
  - execution-time queries/introspection
  - tools
- We don’t do this in other cases
  - “To make this expensive system call, you must use special syntax.”
  - “To load this variable from memory to register, you must decorate it.”
- Exascale only makes it more crucial…

Exascale and Visible Communication

In an exascale world, distinctions between "local" and "remote" get fuzzy

- we need to focus less on "communication" and more on "data transfer"
  - whether between nodes, chips, heterogeneous resources, or…
- even today, NUMA-aware stream can beat NUMA-oblivious by ~2x
  - should we require syntax for cross-NUMA accesses as well?
  - analogy with memory vs. cache vs. registers today
- quandary: which data transfers should require syntax vs. not?
  - personally, I don't think we can afford to decorate every transfer at exascale
  - and think we should be consistent…

#3: Invisible Communication Challenge

Your Challenge: Create PGAS languages that support syntactically invisible communication / data transfers.

Bonus: possibility of having cake and eating it too:

- Models in which the programmer could "opt in" to syntactic model
  - e.g., "compiler, force me to identify all data transfers within this code block"
    - (the "const" of PGAS programming?)
  - e.g., "This message passing / library call implies communication."

#4: Support User-Extensible Parallel Policies

Today's Motivation for User-Extensibility

Questions like the following need to become obsolete:

- "Are arrays in your language row- or column major order? Or …?"
- "How are arrays in your language distributed across memories?"
- "How does your language schedule parallel loop iterations?"

…The only correct answer should be "whatever you want."

- For a given architecture, there is rarely an optimal choice
  - It is highly algorithm- / setting-dependent
- It also seems unlikely that a compiler will automatically make the right/ best choices for us
- Thus, programmers need the ability to control, specify, and abstract away such crucial policy decisions

Exascale's Motivation for User-extensibility

With exascale, so much is in flux…

- system architectures
- their abstract machine models
- algorithms and applications

…that we should be even more skittish about hard-coding such policies into languages

- we need to be adaptable to whatever comes
- we need to be able to experiment
- we can't require algorithms to be rewritten at lowest levels each time

So, user-extensible policies seem even more critical

#4: User-Defined Policy Challenge

Your Challenge: Design PGAS languages that permit crucial policies defining how a parallel algorithm is mapped to the system to be defined by end-users.

#5: Appeal to Conventional Programmers

Attracting Conventional Programmers

- **HPC is a reasonably small sector of computer science**
  - one with an overarching focus on hardware/speed over software
  - we also have trouble attracting / retaining young / talented developers
- **A language that appeals to mainstream programmers would be a boon to the field**
  - enable an open-source movement that would not be possible in HPC
  - lower barriers to working within HPC
  - broaden the market for HPC systems
    - ("every programmer a potential HPC programmer")
- **Consider Python's rapid rise in popularity and use**

Why has Python enjoyed such success?

- **Attractive language design**
  - readable/writable/intuitive
  - successful in spite of some arguably unattractive choices
- **Flexible, general-purpose**
- **Good interoperability support**
- **Rich, extensible library**
- **Open to community code / design contributions**
- **Mechanism for trivially downloading code packages**
- **Good, online documentation**
- **(and almost certainly other things I'm overlooking…)**

#5: Conventional Programmer Challenge

Your Challenge: Become the Python of (scalable) parallel computing (doesn't mean you need to be Python per se)

Summary

What PGAS needs to do for exascale / general adoption:

1. Move beyond SPMD programming
2. Support locality distinctly from parallelism
3. Support communication without syntax
4. Support user-extensible parallel policies
5. Appeal to conventional programmers

Questions for the audience:

- What do you think I'm missing?
- What do you think I've got wrong?

("Why does he keep saying 'your challenge' rather than 'our challenge'?)

("Hey, we're [mm:ss] into the talk and Brad hasn't mentioned Chapel yet… What's up with that?!?")

A: I'm saying "your challenge" because I believe Chapel is already well-positioned for exascale/adoption.

What is Chapel?

- **An emerging parallel programming language**
  - Design and development led by Cray Inc.
    - in an open-source manner at GitHub
    - in collaboration with academia, labs, industry; domestically & internationally
- **Goal: Improve productivity of parallel programming**
- **A work-in-progress**

1. Chapel is a post-SPMD Language

- **Chapel supports a rich set of parallelism styles:**
  - ✓ divide-and-conquer
  - ✓ producer-consumer
  - ✓ streaming / pipeline parallelism
  - ✓ data parallelism
  - ✓ SPMD (including MPI calls)
  - ✓ arbitrary compositions of the above
- **And supports diverse HW and SW parallelism:**

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

* = not yet production-grade

2. Locality and Parallelism are Orthogonal in Chapel

- This is a **parallel**, but local program:

```
begin writeln("Hello world!");
writeln("Goodbye!");
```

- This is a **distributed**, but serial program:

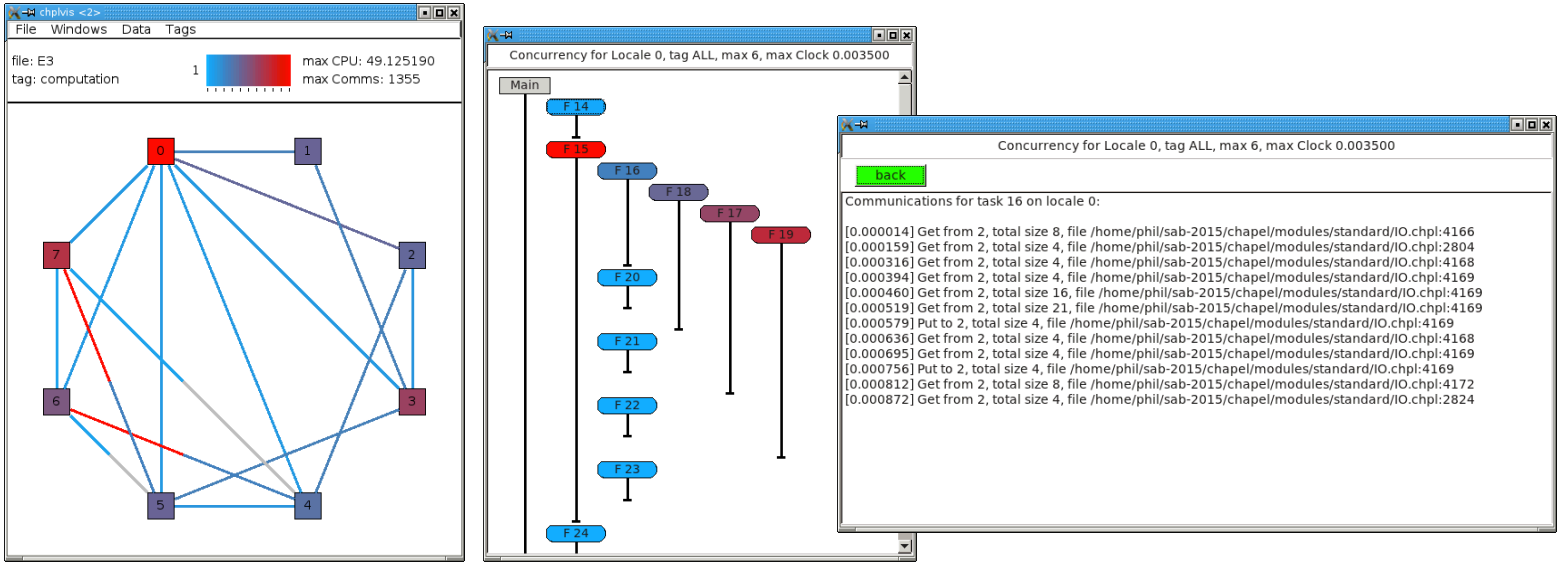
```
writeln("Hello from locale 0!");
on Locales[1] do writeln("Hello from locale 1!");
writeln("Goodbye from locale 0!");
```

- This is a **distributed** and **parallel** program:

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

3. Communication is Invisible in Chapel

- **Yet, users can still reason about data transfers**
  - semantic model is explicit about where data is placed / tasks execute
  - execution-time queries support reasoning about locality
    - `here` – where is this task running?
    - `x.locale` – where is x stored?
  - new chplvis tool supports visualization of communication
    - developed by Phil Nelson, WWU



4. Chapel's parallel features are user-extensible

Chapel users can specify…

…layouts and distributions of arrays

- how are elements distributed across memories? (if they are)
- how are elements laid out in local memories?

…parallel schedules of parallel loops

- how many tasks are applied and where do they execute?
- how are iterations divided between the tasks?

…mapping of Chapel to target architectures

- what is the (abstract) structure of the compute nodes?
- how are variables allocated within a given (sub)locale?
- how are tasks executed within a given (sub)locale?

…and do so by writing Chapel code, not compiler mods

- In addition, Chapel's "built-in" features use this same framework

5. Chapel appeals to conventional users

- **non-HPC programmers are finding Chapel attractive**
  - as are HPC programmers…
  - see CHIUW 2015 talks & state-of-the-project slides for evidence

- **Python enthusiasts make positive comparisons to Chapel**
  - but we still have a lot to learn from Python about deployment and community-building

Chapel Summary

- ☑ 1. Move beyond SPMD programming
- ☑ 2. Support locality distinctly from parallelism
- ☑ 3. Support communication without syntax
- ☑ 4. Support user-extensible parallel policies
- ☐ 5. Appeal to conventional programmers

With Chapel, we made a number of long-game design decisions that we believe make it well-positioned to be a successful exascale PGAS language.

The Chapel Team at Cray (spring 2015)

Note: We currently have two positions open if you or a colleague are looking…

Homework

Read: "HPC is Dying and MPI is Killing it" blog article

Jonathan Dursi, April 3, 2015

Pictured: The HPC community bravely holds off the incoming tide of new technologies and applications. Via the BBC.

Homework

Critical thinking exercises for PGAS:

- What are we doing that feels lame if we truly want to be successful?
  - where “we” could be “your PGAS group” as well as “PGAS as a whole”

- What is a healthy # of PGAS languages?
  - At what point does “strength in numbers” become “ignored by association” or “sufficiently imprecise to not add value”?

Closing PGAS Message: The Inevitability of an Adopted PGAS Language

- **I believe that the broad adoption of a PGAS language is merely a matter of "when" and "which", not "whether"**
  - Most programmers don't write much assembly
  - Most use MPI rather than targeting lower-level network interfaces
    ⇒ programmers are willing to give up control, performance for benefits
  - More and more non-HPC people want to do parallel programming
    - and they won't be as willing to suffer as our programmers are

⇒ A better way of parallel programming is inevitable

And, it is almost certain to be PGAS in nature

- (Because what else would it be to gain broad adoption?)
- Note: this doesn't mean its developers will come from this community or necessarily think of it as PGAS
- But…wouldn't it be great if it did come from our community?

Closing Chapel Messages

The Chapel project is making good progress and gaining mindshare, and we're currently hiring.

Come join us for the SC15 Chapel Lightning Talks BoF or CHIUW 2016.

(Or, kick off a Chapel collaboration and submit a talk to one of these events).

Chapel Resources

Chapel Websites

Project page: http://chapel.cray.com

- overview, papers, presentations, language spec, …

GitHub page: https://github.com/chapel-lang

- download Chapel; browse source repository; contribute code

Facebook page: https://www.facebook.com/ChapelLanguage

Suggested Chapel Reading

Overview Papers:

- *A Brief Overview of Chapel*, Chamberlain (early draft of a chapter for *A Brief Overview of Parallel Programming Models*, edited by Pavan Balaji, to be published by MIT Press in 2015).
  - *a detailed overview of Chapel's history, motivating themes, features*

- *The State of the Chapel Union* [slides], Chamberlain, Choi, Dumler, Hildebrandt, Iten, Litvinov, Titus. CUG 2013, May 2013.
  - *a higher-level overview of the project, summarizing the HPCS period*

Lighter Chapel Reading

Blog Articles:

- *Chapel: Productive Parallel Programming*, Cray Blog, May 2013.
  - *a short-and-sweet introduction to Chapel*
- *Why Chapel?* (part 1, part 2, part 3), Cray Blog, June-October 2014.
  - *a series of articles answering common questions about why we are pursuing Chapel in spite of the inherent challenges*
- *Six Ways to Say "Hello" in Chapel* (part 1), Cray Blog, September 2015.
  - *a series of articles to illustrate the basics of Chapel*
- *[Ten] Myths About Scalable Programming Languages*, IEEE TCSC Blog (index available on chapel.cray.com "blog articles" page), April-November 2012.
  - *a series of technical opinion pieces designed to combat standard arguments against the development of high-level parallel languages*

Chapel Community Resources

SourceForge page: https://sourceforge.net/projects/chapel/

- hosts community mailing lists
  (also serves as an alternate release download site to GitHub)

Mailing Aliases:

write-only:

- chapel_info@cray.com: contact the team at Cray

read-only:

- chapel-announce@lists.sourceforge.net: read-only announcement list

read/write:

- chapel-users@lists.sourceforge.net: user-oriented discussion list
- chapel-developers@lists.sourceforge.net: developer discussion
- chapel-education@lists.sourceforge.net: educator discussion
- chapel-bugs@lists.sourceforge.net: public bug forum

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.

CRAY
THE SUPERCOMPUTER COMPANY

http://chapel.cray.com chapel_info@cray.com http://sourceforge.net/projects/chapel/

Backup Slides

PGAS: What's in a Name?

		memory model	*programming model*	*execution model*	*data structures*	*communication*
	MPI	distributed memory	cooperating executables (often SPMD in practice)		manually fragmented	APIs
	OpenMP	shared memory	global-view parallelism	shared memory multithreaded	shared memory arrays	N/A
Trad. PGAS Languages	**CAF**	PGAS	Single Program, Multiple Data (SPMD)		co-arrays	co-array refs
	UPC				1D block-cyc arrays/ distributed pointers	implicit
	Titanium				class-based arrays/ distributed pointers	method-based
	Chapel	PGAS	global-view parallelism	distributed memory multithreaded	global-view distributed arrays	implicit

(Saying a language is PGAS says very little about it in the grand scheme of things)