

Exascale: Your Opportunity to Create a Decent HPC Language

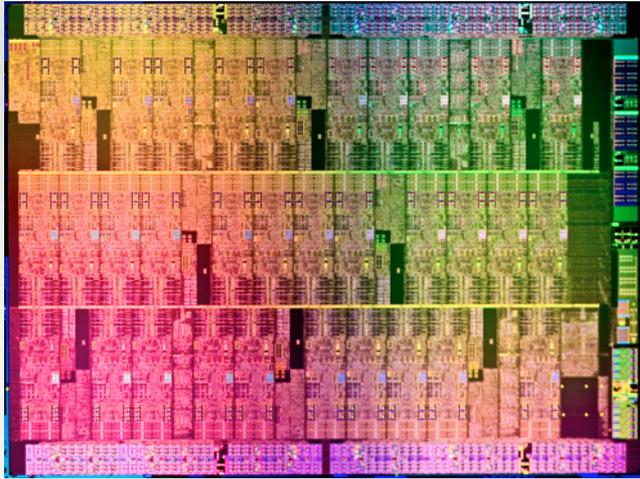
Brad Chamberlain, Cray Inc.

PPME: Productive Programming Models for Exascale

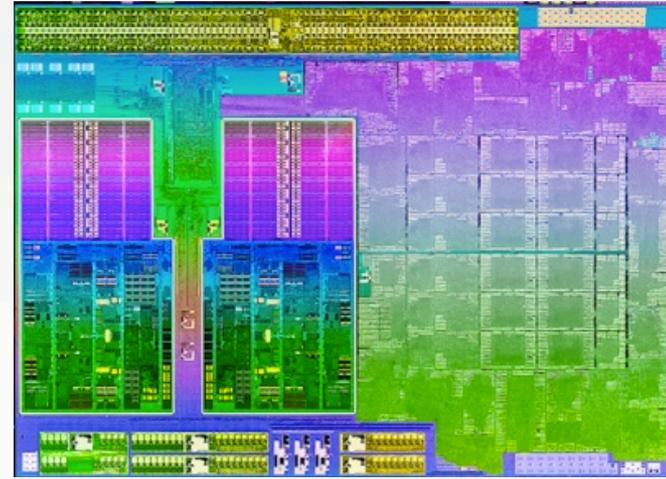
August 14th, 2012



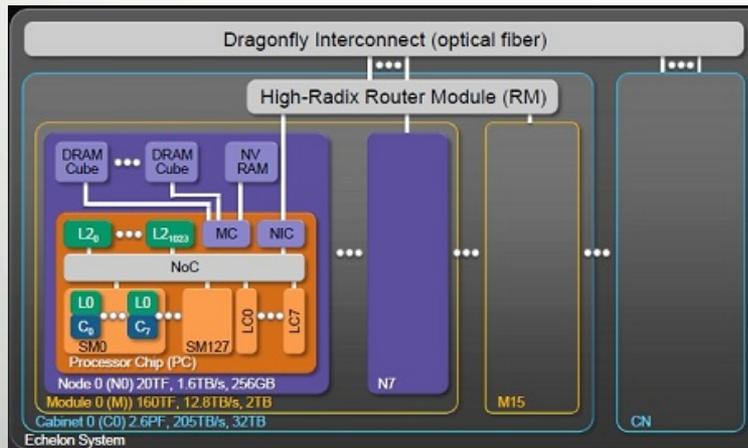
Prototypical Exascale Processor Technologies



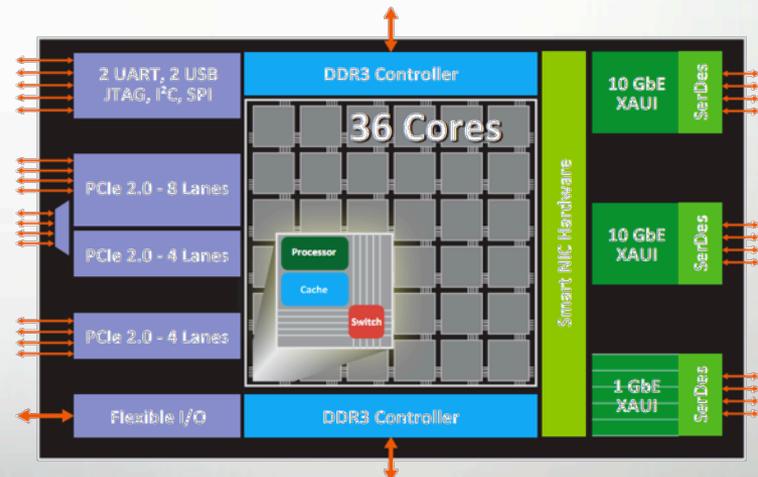
Intel MIC



AMD Trinity

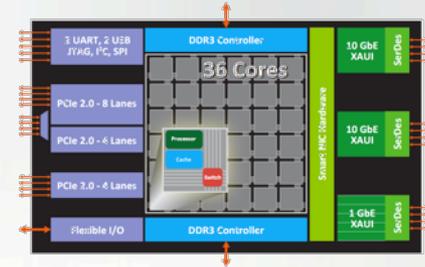
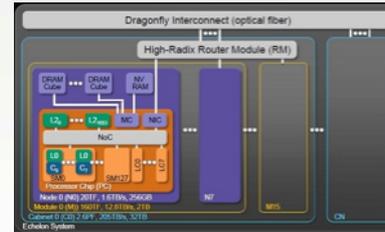
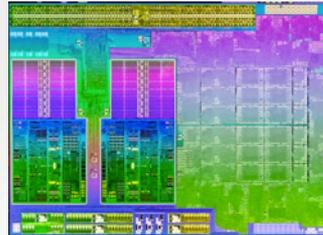
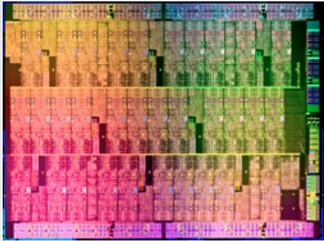


Nvidia Echelon



Tilera Tile-Gx

General Characteristics of These Architectures



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

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

Exascale Challenges

- intra-node locality and heterogeneity concerns
- limited memory bandwidth, memory::FLOP ratios
- increased resiliency concerns
- increased sensitivity to energy usage
- diversity of abstract machine models (at least initially)
- traditional programming models aren't a good fit

A time to be afraid? (very afraid?)

Or, an opportunity?

(If we have to switch to something new, we may as well take the opportunity to adopt something decent while we're at it)

“But I Don’t Care About Exascale!”

A possible contrarian viewpoint: *“I don’t care about Exascale because...*

...it’s too expensive.”

...it won’t serve a broad enough community.”

...it doesn’t suit my application area.”

...I don’t believe it’s possible/practical/worthwhile.”

...anyone who can afford an Exascale machine can afford to hire an army of programmers to suffer through whatever terrible notations are required to utilize it effectively.”

But look, these challenges are not Exascale-specific...

Exascale Programming Model Wishlist

general parallelism:

data parallelism: to take advantage of SIMD HW units; for simplicity when applicable

task parallelism: to fire asynchronous computations off to accelerators; for data-driven algorithms

varying granularities/nestings: for generality across computations & levels of the machine

locality control: to tune for locality/affinity across the machine (inter- and intra-node)

resilience-/energy-aware features: to deal with emerging issues at system scale

user extensibility: to deal with unknowns in next-generation architectures and algorithms

productivity features: because you know you want them

Productivity: no longer cool?

- Has “productivity” run its course, thanks to HPCS?
 - Maybe programmatically or politically...
 - But look, who wouldn’t want code that’s easier to...
 - ...write
 - ...read
 - ...tune
 - ...maintain
 - ...port
 -

The goal remains worthy even if the term has grown stale

How to attract new parallel programmers?



“Decent” Parallel Languages

- What was the last parallel notation you used that felt
 - productive?
 - high-level?
 - powerful?
 - flexible?
 - effective?
 - modern?
 - fun?
 - (...all the things we judge good software by...)?



(Because that's what we're competing with)

Exascale Scorecard for HPC Programming Models

	Fortran	C/C++	MPI	OpenMP	UPC
data parallelism					
task parallelism					
parallel nesting/granularities					
locality control					
resilience/energy-awareness					
user-extensibility					
productivity					

Exascale Scorecard for HPC Programming Models

	Fortran	C/C++	MPI	OpenMP	UPC
data parallelism	~			~	~
task parallelism				~	
parallel nesting/granularities				~	
locality control			~		~
resilience/energy-awareness			~		
user-extensibility					
productivity				~	

Exascale Scorecard for HPC Programming Models

	Fortran	C/C++	MPI	OpenMP	UPC
data parallelism	~	X	X	~	~
task parallelism	X	X	X	~	X
parallel nesting/granularities	X	X	X	~	X
locality control	X	X	~	X	~
resilience/energy-awareness	X	X	~	X	X
user-extensibility	X	X	X	X	X
productivity	X	X	X	~	X

Programming Exascale Architectures

Q: Are we ready?

A: In a nutshell, no

Q: Why?

A: We've built too many assumptions about our target architectures into our programming models

- granularity and style of parallelism
- mode of communication
- single level of locality, if any at all
- inflexible, non-productive base languages

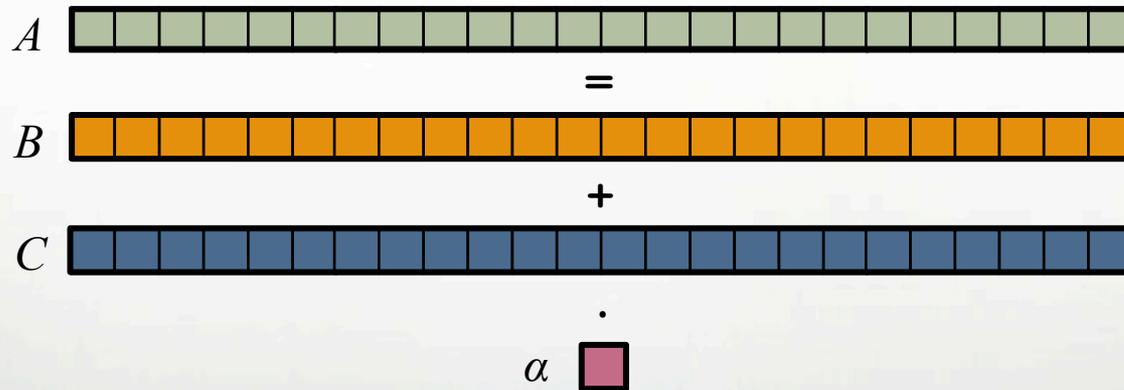
productive programming models should be focused more on the user's application than on the hardware

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:

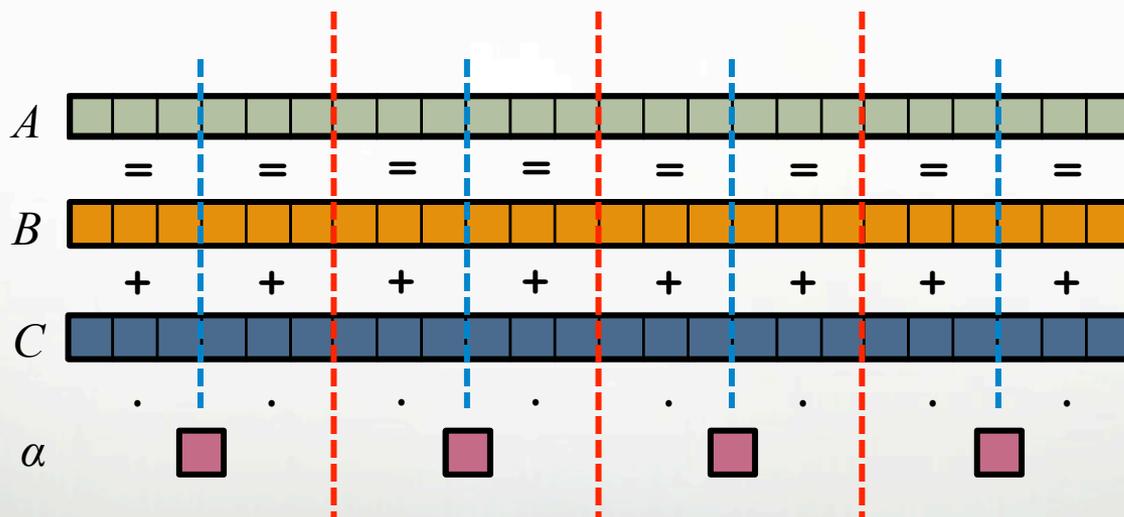


STREAM Triad: a trivial parallel computation

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 (distributed memory multicore):



STREAM Triad: MPI

MPI

```
#include <hpcc.h>

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

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

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

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

    return errCount;
}

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

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

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

```
    if (!a || !b || !c) {
        if (c) HPCC_free(c);
        if (b) HPCC_free(b);
        if (a) HPCC_free(a);
        if (doIO) {
            fprintf( outFile, "Failed to allocate memory (%d).
\n", VectorSize );
            fclose( outFile );
        }
        return 1;
    }

    for (j=0; j<VectorSize; j++) {
        b[j] = 2.0;
        c[j] = 0.0;
    }

    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

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] = 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;
}

```

STREAM Triad: MPI+OpenMP vs. CUDA

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_MALLOC( 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;
}
```

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

Hybrid solutions are our only recourse to program Exascale at all; and while they're reasonable as conservative approaches, they're far from ideal

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);
    if( N % dimBlock.x != 0 ) dimGrid.x+=1;

    set_array<<<dimGrid,dimBlock>>>(d_b, 2.5f, N);

    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];
}
```

By Analogy: Let's Cross the United States!



By Analogy: Let's Cross the United States!



...Hey, what's that sound?

STREAM Triad: Chapel

MPI + OpenMP

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

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

int HPCC_StarStream(HPCC_Params *par
    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 == myR
    MPI_Reduce( &rv, &errCount, 1, MPI

    return errCount;
}

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

    VectorSize = HPCC_LocalVectorSize(
    a = HPCC_XMALLOC( double, VectorSi
    b = HPCC_XMALLOC( double, VectorSi
    c = HPCC_XMALLOC( double, VectorSi

    if (!a || !b || !c) {
        if (c) HPCC_free(c);
        if (b) HPCC_free(b);
        if (a) HPCC_free(a);
        if (doIO) {
```

Chapel

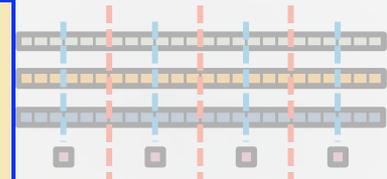
```
config const m = 1000,
              alpha = 3.0;

const ProblemSpace = [1..m] dmapped ...;

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

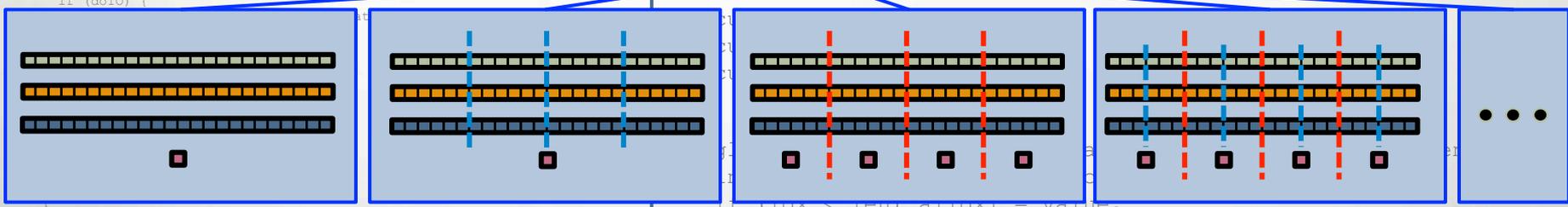
B = 2.0;
C = 3.0;

A = B + alpha * C;
```



```
;
;
;
N);
N);
_c, d_a, scalar, N);
```

the special sauce



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

```
scalar =
#ifdef _OPE
#pragma omp
#endif
for (j=0;
    a[j] =
    HPCC_free
    HPCC_free
    HPCC_free
return 0;
}
```

A Common Question

Q: Didn't we try this before with HPF?

Q': Orville, didn't Percy Pilcher die in *his* prototype powered aircraft?



A': No Wilbur, he died in a glider; and even if it had been in his prototype, that doesn't mean we're doomed to fail.

Q: How Can Chapel Succeed When HPF Failed?

A: Chapel has had the chance to learn from HPF's mistakes (and other languages' successes and failures)

- Why did HPF fail?
 - lack of sufficient performance soon enough
 - vagueness in execution/implementation model
 - only supported a single level of data parallelism, no task/nested
 - inability to drop to lower levels of control
 - fixed set of limited distributions on dense arrays
 - lacked richer data parallel abstractions
 - lacked an open source implementation
 - too Fortran-based for modern programmers
 - ...?
- The failure of one language---even a well-funded, US-backed one---does not dictate the failure of all future languages

(For more on this topic see https://www.ieeetcsc.org/activities/blog/myths_about_scalable_parallel_programming_languages_part2)

Chapel in a Nutshell

Chapel: a parallel language that has emerged from DARPA HPCS

- **general parallelism:**
 - data-, task-, and nested parallelism
 - highly dynamic multithreading or static SPMD-style
- **multiresolution philosophy:** high-level features built on low-level
 - to provide “manual overrides”
 - to support a separation of concerns (application vs. parallel experts)
- **locality control:**
 - explicit or data-driven placement of data and tasks
 - locality expressed distinctly from parallelism
- **features for productivity:** type inference, iterators, rich array types
- **portable:** designed and implemented to support diverse systems
- **open source:** developed and distributed under the BSD license
- **plausibly adoptable:** forward-thinking HPC users want a mature version

Chapel: Well-Positioned for Exascale

data parallelism	✓
task parallelism	✓
parallel nesting/granularities	✓
locality control	~
resilience/energy-awareness	X
user-extensibility	✓
productivity	✓

Chapel: Limitations for Exascale

Chapel limitations for Exascale, today:

- locales only support a single level of hierarchy
 - useful for horizontal (inter-node) locality
 - less so for describing additional hierarchy within a node
- lack of fault tolerance/error handling features

In Chapel's original design, these were both considered "version 2.0" features due to...

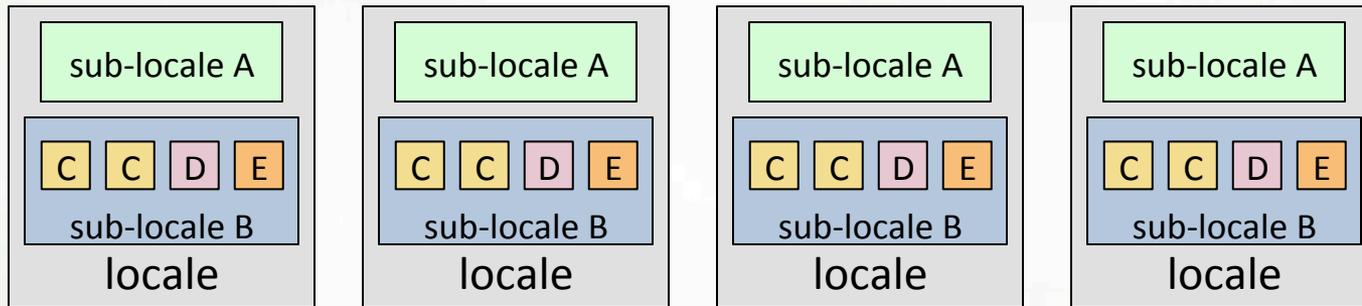
...our focus on petascale systems within HPCS

...the knowledge that our plate was already quite full

Current Work: Hierarchical Locales

Concept:

- Support locales within locales to describe architectural sub-structures within a node



- As with traditional locales, *on-clauses* and *domain maps* can be used to map tasks and variables to a sub-locale's memory and processors
- Locale structure is defined as Chapel code
 - permits implementation policy to be specified in-language
 - introduces a new Chapel role: *architectural modeler*

Sublocale Examples: Tiled Processors

```

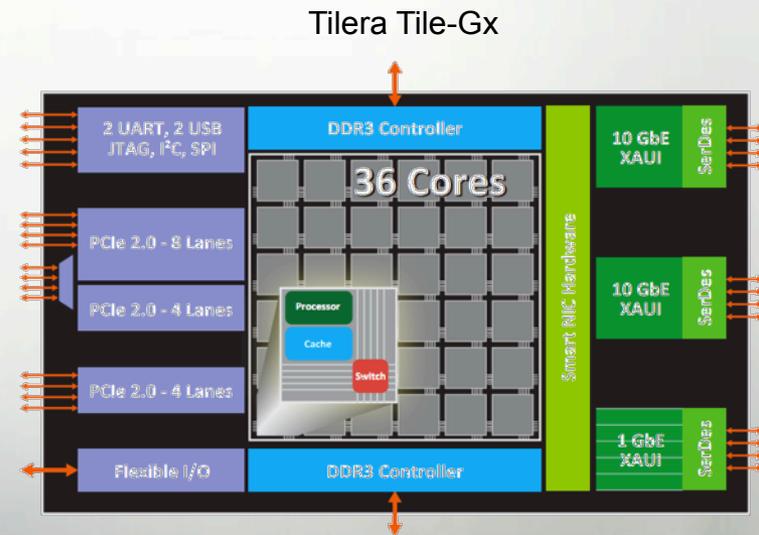
class locale: AbstractLocale {
    const xt = 6, yt = xTiles;
    const sublocGrid: [0..#xt, 0..#yt] tiledLoc = ...;
    const allSublocs: [0..#xt*yt] tiledLoc = ...;
    // tasking interface
    // memory interface
}

```

```

class tiledLoc: AbstractLocale {
    // tasking interface
    // memory interface
}

```



Sublocale Examples: Hybrid Processors

```

class locale: AbstractLocale {
  const numCPUs = 2, numGPUs = 2;
  const cpus: [0..#numCPUs] cpuLoc = ...;
  const gpus: [0..#numGPUs] gpuLoc = ...;
  // tasking interface
  // memory interface
}

```

```

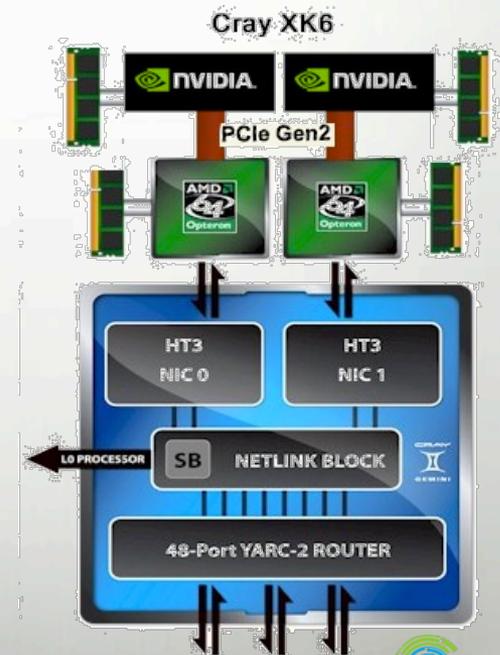
class cpuLoc: AbstractLocale { ... }

```

```

class gpuLoc: AbstractLocale {
  // sublocales for different
  // memory types, thread blocks...?
  // tasking, memory interfaces
}

```



“How can I help Chapel?”

- Evolve from “How will they do it?” to “How will we do it?”
- Let our management (and yours) know what you think about it
- Help find funding to pursue Exascale-/DOE-specific challenges
- Help us tap into the mainstream/open-source community
- Become a collaborator who is truly interested in pitching in
 - Computer scientists with mutually beneficial technologies
 - Industry partners
 - Application experts interested in pair programming studies

Sample Pair-Programming Study: LLNL/LULESH

Apr 2011: LLNL expresses interest in Chapel at Salishan

- made us aware of the LULESH benchmark from DARPA UHPC

Summer 2011: Cray intern ports LULESH to Chapel

- *caveat:* used structured mesh to represent data arrays

Nov 2011: Chapel team tunes LULESH for single-node performance

Dec 2011: Chapel team visits LLNL (talk, tutorial, 1-on-1 sessions)

Mar 2012: Jeff Keasler (LLNL) visits Cray to pair-program

- in one afternoon, converted from structured to unstructured mesh
- impact on code minimal (mostly in declarations) due to:
 - domains/arrays/iterators
 - rank-independent features

Apr 2012: LLNL reports on collaboration at Salishan

Apr 2012: Chapel 1.5.0 release includes current version of LULESH

Next steps: distributed sparse domains, improved scalability

A Parting Question

Q: If, as a community, we/you were to do the audacious and produce a productive/decent parallel language, what would that process look like?

Summary

Productivity continues to be worth striving for

- to improve the lives of existing programmers, especially as the machines get more complex
- to improve our chances of attracting new users

Current programming models aren't ideal for Exascale

- and they will be difficult to fix
- however, that's not to say we should just abandon them

Past failures do not dictate future ones

Exascale represents an opportunity to move to more ideal programming models

- more productive, higher-level, machine-independent, ...

The Chapel Team (Summer 2012)



For More Information

Chapel project page: <http://chapel.cray.com>

- overview, papers, presentations, language spec, ...

Chapel SourceForge page: <https://sourceforge.net/projects/chapel/>

- release downloads, public mailing lists, code repository, ...

Blog Series (for more opinions like these):

Myths About Scalable Programming Languages:

<https://www.ieeetcsc.org/activities/blog/>

Upcoming Events:

PGAS/PGAS-X (talks): October 10th-12th

SC12 (tutorial, BoFs?): November 12th-16th

