Chapel: Global HPCC Benchmarks and Status Update

Brad Chamberlain
Chapel Team

CUG 2007
May 7, 2007
Chapel

**Chapel**: a new parallel language being developed by Cray

- **Themes:**
  - general parallelism
    - data-, task-, nested parallelism using *global-view* abstractions
    - general parallel architectures
  - locality control
    - data distribution
    - task placement (typically data-driven)
  - narrow gap between mainstream and parallel languages
    - object-oriented programming (OOP)
    - type inference and generic programming
Chapel’s Setting: HPCS

- **HPCS**: High *Productivity* Computing Systems
  - **Goal**: Raise productivity by $10 \times$ for the year 2010
  - **Productivity** = Performance  
    + Programmability  
    + Portability  
    + Robustness

- **Phase II**: Cray, IBM, Sun (July 2003 – June 2006)
  - Evaluation of the entire system architecture’s impact on productivity…
    - processors, memory, network, I/O, OS, runtime, compilers, tools, …
    - ...and new languages:
      - **IBM**: X10  
      - **Sun**: Fortress  
      - **Cray**: Chapel

- **Phase III**: Cray, IBM (July 2006 – 2010)
  - Implement the systems and technologies resulting from phase II
Chapel and Productivity

- Chapel’s Productivity Goals:
  - vastly improve **programmability** over current languages/models
    - writing parallel codes
    - reading, modifying, maintaining, tuning them
  - support **performance** at least as good as MPI
    - competitive with MPI on generic clusters
    - better than MPI on more productive architectures like Cray’s
  - improve **portability** compared to current languages/models
    - as ubiquitous as MPI, but with fewer architectural assumptions
    - more portable than OpenMP, UPC, CAF, …
  - improve **code robustness** via improved semantics and concepts
    - eliminate common error cases altogether
    - better abstractions to help avoid other errors
Outline

✓ Chapel Overview

➢ HPC Challenge Benchmarks in Chapel
  • STREAM Triad
  • Random Access
  • 1D FFT

☐ Project Status and User Activities
HPC Challenge Overview

**Motivation:** Growing realization that top-500 often fails to reflect practical/sustained performance
- measured using HPL, which essentially measures peak FLOP rate
- user applications often constrained by memory, network, …

**HPC Challenge (HPCC):**
- suite of 7 benchmarks to measure various system characteristics
- annual competition based on 4 of the HPCC benchmarks
  - **class 1:** best performance (award per benchmark)
  - **class 2:** most productive
    - 50% performance
    - 50% code elegance, size, clarity

For more information:
- HPCC Competition: [http://www.hpcchallenge.org](http://www.hpcchallenge.org)
STREAM Triad
Introduction to STREAM Triad

**Given:** $m$-element vectors $A, B, C$

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

**Pictorially:**

\[ A \quad B \quad C \]
\[ = \]
\[ + \]
\[ \alpha \]

[Diagram showing the computation process]
Introduction to STREAM Triad

**Given:** \( m \)-element vectors \( A, B, C \)

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

Pictorially (in parallel):
STREAM Triad: Some Declarations

```plaintext
const m = computeProblemSize(elemType, numVectors),
alpha = 3.0;
```
STREAM Triad: Some Declarations

\[
\text{const } m = \text{computeProblemSize}(\text{elementType, numVectors}), \\
\text{alpha} = 3.0;
\]

Chapel Variable Declarations

{ \text{var | const | param} } \ <\text{name}>[: <\text{definition}>] [= <\text{initializer}>]

- \text{var} \Rightarrow \text{can change values}
- \text{const} \Rightarrow \text{a run-time constant (can't change values after initialization)}
- \text{param} \Rightarrow \text{a compile-time constant}

May omit definition or initializer, but not both
- If definition omitted, type inferred from initializer
- If initializer omitted, variable initialized using type's default value

Here, \( m \) has no definition, so its type is inferred using the return type of \text{computeProblemSize()} -- an int
Similarly, \( \text{alpha} \) is inferred to be a real floating point value
STREAM Triad: Some Declarations

```chapel
config const m = computeProblemSize(elemType, numVectors),
    alpha = 3.0;
```

**Configuration Variables**

Preceding a variable declaration with `config` allows it to be initialized on the command-line, overriding its default initializer.

- `config const/var` ⇒ can be overridden on executable command-line
- `config param` ⇒ can be overridden on compiler command-line

```
prompt> stream --m=10000 --alpha=3.14159265
```
STREAM Triad: Core Computation

```
const ProblemSpace: domain(1) distributed(Block) = [1..m];
var A, B, C: [ProblemSpace] elemType;

A = B + alpha * C;
```
STREAM Triad: Core Computation

```plaintext
const ProblemSpace: domain(1) distributed(Block) = [1..m];
var A, B, C: [ProblemSpace] elemType;

Declare a domain

domain: a first-class index set, potentially distributed
(think of it as the size and shape of an array)

domain(1) ⇒ 1D arithmetic domain, indices are integers

A = [1..m] ⇒ a 1D arithmetic domain literal defining the index set:
{1, 2, ..., m}
```

ProblemSpace

1         m
STREAM Triad: Core Computation

```
const ProblemSpace: domain(1) distributed(Block) = [1..m];
var A, B, C: [ProblemSpace] elemType;
```

Specify the domain’s distribution

distribution: describes how to map the domain indices to locales, and how to implement domains (and their arrays)

distributed(Block) ⇒ break the indices into numLocales consecutive blocks

![Diagram of ProblemSpace distribution](image)
STREAM Triad: Core Computation

```chapel
const ProblemSpace: domain(1) distributed(Block) = [1..m];
var A, B, C: [ProblemSpace] elemType;

Declare arrays
arrays: mappings from domains (index sets) to variables. Several flavors:
• dense and sparse rectilinear (indexed by integer tuples)
• associative arrays (indexed by value types)
• opaque arrays (indexed anonymously to represent sets & graphs)

ProblemSpace

A

B

C
```
STREAM Triad: Core Computation

**Expressing the computation**

*whole-array operations*: support standard scalar operations on arrays in an element-wise manner

\[
\begin{align*}
A & = \text{[Array]} \\
B & = \text{[Array]} \\
C & = \text{[Array]} \\
\alpha & = \text{[Scalar]}
\end{align*}
\]

\[
A = B + \alpha \times C;
\]
STREAM Triad: Core Computation

```chapel
const ProblemSpace: domain(1) distributed(Block) = [1..m];
var A, B, C: [ProblemSpace] elemType;

A = B + alpha * C;
```
Random Access
Introduction to Random Access

Given: $m$-element table $T$ (where $m = 2^n$ and initially $T_i = i$)
Compute: $N_U$ random updates to the table using bitwise-xor
Pictorially:
Introduction to Random Access

Given: \( m \)-element table \( T \) (where \( m = 2^n \) and initially \( T_i = i \))
Compute: \( N_U \) random updates to the table using bitwise-xor

Pictorially:
Introduction to Random Access

Given: \( m \)-element table \( T \) (where \( m = 2^n \) and initially \( T_i = i \))
Compute: \( N_U \) random updates to the table using bitwise-xor

Pictorially:

= 21 \( \Rightarrow \) xor the value 21 into \( T_{(21 \mod m)} \)

repeat \( N_U \) times
Introduction to Random Access

Given: $m$-element table $T$ (where $m = 2^n$ and initially $T_i = i$)
Compute: $N_U$ random updates to the table using bitwise-xor

Pictorially (in parallel):
Introduction to Random Access

Given: \( m \)-element table \( T \) (where \( m = 2^n \) and initially \( T_i = i \))

Compute: \( N_U \) random updates to the table using bitwise-xor

Pictorially (in parallel):

Random Numbers
Not actually generated using lotto ping-pong balls!
Instead, implement a pseudo-random stream:
  • \( k \)th random value can be generated at some cost
  • given the \( k \)th random value, can generate the \((k+1)\)st much more cheaply
Random Access: Domains and Arrays

\[
\text{const TableSpace: domain(1) distributed(Block) = [0..m);}
\]
\[
\text{var T: [TableSpace] elemType;}
\]
\[
\text{const UpdateSpace: domain(1) distributed(Block) = [0..N_U);}
\]
Random Access: Random Value Iterator

```chapel
iterator RAStream(block) {
    var val = getNthRandom(block.low);
    for i in block {
        getNextRandom(val);
        yield val;
    }
}

def getNthRandom(in n) { ... }

def getNextRandom(inout x) { ... }
```
Random Access: Random Value Iterator

```
iterator RAStream(block) {
    var val = getNthRandom(block.low);
    for i in block {
        getNextRandom(val);
        yield val;
    }
}
```

**Defining an iterator**

- **iterator**: similar to a function but generates a stream of return values; invoked using `for` and `forall` loops
- **yield**: like a return statement but the iterator’s execution continues logically after returning the value
- **RAStream()**: an iterator that generates a random value for each index in `block`

E.g., to iterate over the entire stream sequentially, one could use:
```
for r in RAStream([0..N_U)) { ... }
```
Random Access: Random Value Iterator

iterator RAStream(block) {
  var val = getNthRandom(block.low);
  for i in block {
    getNextRandom(val);
    yield val;
  }
}

def getNthRandom(in n) { ... }

def getNextRandom(inout x) { ... }
Random Access: Computation

\[ \text{[i in TableSpace] T(i) = i;} \]

\[ \text{forall block in UpdateSpace.subBlocks do} \]
\[ \text{for r in RAStream(block) do} \]
\[ \text{T(r & indexMask) ^= r;} \]
Random Access: Computation

\[ i \text{ in TableSpace} \] \[ T(i) = i; \]

\textbf{Initialization}

Uses \textit{forall expression} to initialize table

\textbf{Computing the Updates}

Express table updates by invoking iterators:

\textit{subBlocks}: a standard iterator that generates blocks of indices appropriate for the target machine’s parallelism

\textit{RAStream()}: our iterator for generating random values

Effectively: generate parallel chunks of work; iterate over chunks serially performing updates
Random Access: Computation

\[ [i \text{ in } \text{TableSpace}] \ T(i) = i; \]

\[
\text{forall block in UpdateSpace.subBlocks do for } r \text{ in RASTream(block) do}
\]
\[
T(r \& \text{indexMask}) ^= r;
\]
Introduction to FFT

Given: \(m\)-element vector \(z\) of complex numbers (where \(m = 2^n\))
Compute: 1D Discrete Fourier Transform of \(z\)
Pictorially (using a radix-4 algorithm):
FFT: Computation

```chapel
for i in [2..log2(numElements)) by 2 {
    const m = radix*span, m2 = 2*m;

    forall (k, k1) in (Adom by m2, 0..) {
        var wk2 = ..., wk1 = ..., wk3 = ...;

        forall j in [k..k+span) do
            butterfly(wk1, wk2, wk3, A[j..j+3*span by span]);

        wk1 = ...; wk3 = ...; wk2 *= 1.0i;

        forall j in [k+m..k+m+span) do
            butterfly(wk1, wk2, wk3, A[j..j+3*span by span]);
    }
    span *= radix;
}

def butterfly(wk1, wk2, wk3, inout A:[1..radix]) { ... }
```
FFT: Computation

for i in [2..log2(numElements)) by 2 {
    const m = radix*span, m2 = 2*m;

    forall (k,k1) in (Adom by m2, 0..) {
        var wk2 = …, wk1 = …, wk3 = …;
        forall j in [k..k+span) do
            butterfly(wk1, wk2, wk3, A[j..j+3*span by span]);
        wk1 = …; wk3 = …; wk2 *= 1.0i;
        forall j in [k+m..k+m+span) do
            butterfly(wk1, wk2, wk3, A[j..j+3*span by span]);
    }
    span *= radix;
}

def butterfly(wk1, wk2, wk3, inout A:[1..radix]) { … }

Sequential loop to express phases of computation

Nested forall loops to express a phase’s parallel butterflies

Support for complex and imaginary types simplifies math

Generic arguments allow butterfly() to be called with complex, real, or imaginary twiddle factors
FFT: Computation

```chapel
for i in [2..log2(numElements)) by 2 {
    const m = radix*span, m2 = 2*m;

    forall (k,k1) in (Adom by m2, 0..) {
        var wk2 = ..., wk1 = ..., wk3 = ...;

        forall j in [k..k+span) do
            butterfly(wk1, wk2, wk3, A[j..j+3*span by span]);
        wk1 = ...; wk3 = ...; wk2 *= 1.0i;

        forall j in [k+m..k+m+span) do
            butterfly(wk1, wk2, wk3, A[j..j+3*span by span]);
    }
    span *= radix;
}

def butterfly(wk1, wk2, wk3, inout A:[1..radix]) {
    ...
}
```
HPCC Status, Next Steps

**HPCC Status:**
- all codes compile and run today
- current compiler only targets a single node
- serial performance approaching hand-coded C on a daily basis
- CUG paper…
  - contains full source listings
  - covers codes in more detail
  - describes performance advantages and challenges in Chapel

**What’s Next?**
- demonstrate performance for these codes
  - continue optimizing serial performance
  - add compiler support for targeting multiple nodes
- finish implementing HPL
HPCC Summary

- Chapel supports HPCC codes attractively
  - clear, concise, general
  - parallelism expressed in architecturally-neutral way
  - benefit from Chapel’s global-view parallelism
  - utilizes generic programming and modern SW Engineering principles
  - should serve as an excellent reference for future HPCC competitors

- Note that HPCC benchmarks are relatively simple
  - all data structures are 1D vectors
  - locality very data driven
  - minimal task- & nested parallelism
  - little need for OOP, abstraction

...HPCC designed to stress systems, not languages
  - would like to see similar competitions emerge for richer computations
Outline

✓ Chapel Overview
✓ HPC Challenge Benchmarks in Chapel
  ✓ STREAM Triad
  ✓ Random Access
  ✓ 1D FFT

➢ Project Status and User Activities
Chapel Work

- Chapel Team’s Focus:
  - specify Chapel syntax and semantics
  - implement prototype Chapel compiler
  - code studies of benchmarks, applications, and libraries in Chapel
  - community outreach to inform and learn from users
  - support users evaluating the language
  - refine language based on these activities
Project Status, Next Steps

- **Chapel specification:**
  - revised draft language specification available on Chapel website
  - editing to add additional examples & rationale; improve clarity

- **Compiler implementation:**
  - improving serial performance
  - starting on distributed memory implementation
  - adding missing serial features

- **Code studies:**
  - **NAS Parallel Benchmarks:** CG (well underway), IS, FT, MG
  - **Linear Algebra routines:** block LU, block Cholesky, matrix mult.
  - **Other applications of interest:** Fast Multipole Method, SSCA2, …

- **Release:**
  - made a preliminary release to government team December 2006
  - initial response from those users has been positive, encouraging
  - next release due Summer 2007
Notable User Studies

- Two main efforts to date, both at ORNL:
  - Robert Harrison, Wael Elwasif, David Bernholdt, Aniruddha Shet
    - Fock matrix computations using producer-consumer parallelism
    - coupled model idioms (e.g., for use in CCSM)
  - Richard Barrett, Stephen Poole, Philip Roth
    - stencil idioms: 2D, 3D, sparse
    - sweep3D & wavefront-style computations

- In both cases...
  - great technical discussions and feedback
  - valuable sanity-check for language and implementation
  - studies comparing with Fortress, X10 forthcoming
Chapel Contributors

- **Current:**
  - Brad Chamberlain
  - Steven Deitz
  - Mary Beth Hribar
  - David Iten
  - (Your name here? We’re hiring…)

- **Alumni:**
  - David Callahan
  - Hans Zima (CalTech/JPL)
  - John Plevyak
  - Wayne Wong
  - Shannon Hoffswell
  - Roxana Diaconescu (CalTech)
  - Mark James (JPL)
  - Mackale Joyner (2005 intern, Rice University)
  - Robert Bocchino (2006 intern, UIUC)
For More Information…

BOF today at 4pm

chapel_info@cray.com
bradc@cray.com

http://chapel.cs.washington.edu

Your feedback desired!