* It goes without saying that this somewhat pejorative term doesn’t apply to any of the fine people in this room, all of whom, in my experience, consistently apply the appropriate amount of control to ensure a high degree of quality is reached without succumbing to micromanagement. ** But if you think hard, perhaps you can come up with someone in HPC to whom the label applies? Yeah, like him. Totally. Good call.

** I mean, none of us program exclusively in assembly language any more, right??
Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors
- Static finite element analysis

1 TF – 1998: Cray T3E; 1,024 Processors
- Modeling of metallic magnet atoms

1 PF – 2008: Cray XT5; 150,000 Processors
- Superconductive materials

1 EF – ~2018: Cray ____; ~10,000,000 Processors
- TBD
Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors
• Static finite element analysis
  • Fortran77 + Cray autotasking + vectorization

1 TF – 1998: Cray T3E; 1,024 Processors
• Modeling of metallic magnet atoms
  • Fortran + MPI (?)

1 PF – 2008: Cray XT5; 150,000 Processors
• Superconductive materials
  • C++/Fortran + MPI + vectorization

1 EF – ~2018: Cray ____; ~10,000,000 Processors
• TBD
  • TBD: C/C++/Fortran + MPI + CUDA/OpenCL/OpenMP/??? + ???
HPC has traditionally given users...
...low-level programming models
...ones that are closely related to the underlying hardware

**benefits:** lots of control; decent generality; easy to implement

**downsides:** lots of user-managed detail; brittle to changes

**Thesis:** Lower-level notations constrain implementation options more, for better or worse; for the purposes of exploiting asynchrony and utilizing exascale, mostly “worse.”
int n = computeProbSize(),
    myN = computeMyProbSize(n);
double A[myN], B[myN];
double sumOfSquares, mySumOfSquares;

for (i=0; i<myN; i++)
    mySumOfSquares += A[i]*A[i] + B[i]*B[i];

MPI_Reduce(&mySumOfSquares, &sumOfSquares,
            MPI_SUM, MPI_DOUBLE, 0, MPI_COMM_WORLD);
int n = computeProbSize(),
    myN = computeMyProbSize(n);

double A[myN], B[myN];
double sumOfSquares, mySumOfSquares;

for (i=0; i<myN; i++)
    mySumOfSquares += A[i]*A[i] + B[i]*B[i];

MPI_Reduce(&mySumOfSquares, &sumOfSquares,
    MPI_SUM, MPI_DOUBLE, 0, MPI_COMM_WORLD);

Specified
• **Unit of Parallelism:** Cooperating Executable (via use of MPI)
• **Other Decisions:** Data Decomposition, Local Computation Style, and Synchronous Communication (via program text)
int n = computeProbSize(),
    myN = computeMyProbSize(n);

double A[myN], B[myN];
double sumOfSquares, mySumOfSquares;

for (i=0; i<myN; i++)
    mySumOfSquares += A[i]*A[i] + B[i]*B[i];

MPI_Reduce(&mySumOfSquares, &sumOfSquares,
            MPI_SUM, MPI_DOUBLE, 0, MPI_COMM_WORLD);

Unspecified
• Communication Details: All-to-one? Combining Tree? What arity? Who does each node send to/recv from? (and with good software engineering, we could arguably do more...)
Traditional PGAS Languages: UPC, CAF, Titanium

- Communication expressed as variable accesses
  - says more about *what* should be moved than *how* (or, arguably, *when*)
    - synchronization is decoupled from data transfer
    - admits more asynchronous implementations

- Yet control and data models are still fairly restricted
  - SPMD execution
  - limited support for distributed arrays
Chapel: A Next-Generation PGAS Language

- A new parallel language being developed by Cray Inc. under DARPA HPCS
- a PGAS language, but non-traditional:
  - rich set of arrays: multidimensional, sparse, associative, unstructured
  - explicit concepts for describing locality/affinity
    - e.g., *locale* type represents architectural locality
  - more general/dynamic/multithreaded parallelism
Global-View: Sum of Squares Reduction

```cpp
config const n = computeProblemSize();
const D = [1..n];
var A, B: [D] real;

const sumOfSquares = + reduce (A**2 + B**2);
```
Global-View: What is Specified?

```plaintext
config const n = computeProblemSize();
const D = [1..n];
var A, B: [D] real;

const sumOfSquares = + reduce (A**2 + B**2);
```

Specified

- **Intention**: “We want to compute a sum reduction”
config const n = computeProblemSize();
const D = [1..n);
var A, B: [D] real;

const sumOfSquares = + reduce (A**2 + B**2);

Unspecified
• Local Computation Style: statically partitioned? dynamically? details?
• Communication Details: All-to-one? Combining Tree? What arity? Who does each node send to/recv from?
  • implemented using message passing? puts/gets? active messages?
No reason to believe performance must suffer

“High-level doesn’t necessarily mean slow if your abstractions are designed to map efficiently.”

- Pat Hanrahan (my wording)

```cpp
const sumOfSquares = + reduce (A**2 + B**2);
```

“Just because HPF failed doesn’t mean all high-level languages must.”

- Chamberlain corollary

Another example: sparse arrays

- CSR using 1D arrays ⇒ lots of indirect indexing
- OOP ⇒ lots of field/method indirection
- put sparse arrays in language ⇒ users rejoice; compiler has rich new semantics to reason about & optimize
config const n = computeProblemSize();
const D = [1..n];
var A, B: [D] real;

const sumOfSquares = + reduce (A**2 + B**2);

How do we implement this global-view operation in practice?

**ZPL:** Block-distributed arrays, serial per node, ... (inflexible)

**HPF:** Not particularly well-defined ("trust the compiler")

**Chapel:** Very well-defined and flexible... stay tuned...
config const n = computeProblemSize();
const D = [1..n, 1..n];
var A, B: [D] real;

const sumOfSquares = + reduce (A**2 + B**2);

Computation is Rank-Independent
(and with a bit more work on the user’s part, the declarations could be too)
config const n = computeProblemSize();
const D = [1..n, 1..n];
var A, B: [D] real;

const sumOfSquares = + reduce forall ij in D do
  (A[ij]**2 + B[ij]**2);

// or: forall (a,b) in (A,B) do
//     (a**2 + b**2);

Computation also has rank-independent loop-based forms
config const n = computeProblemSize();
const D = [1..n, 1..n];
var A, B: [D] real;

var sumOfSquares$: sync real;

begin sumOfSquares$ = + reduce forall (a,b) in (A,B) do (a**2 + b**2);
dosomeOtherStuff(...);
...sqrt(sumOfSquares$)...

Fire off asynchronous task, storing result in sync (full/empty) variable
Read of sync variable blocks until result has been written.
“Brad, this all sounds great for urbane programmers like you and me, but I’m really concerned about my poor control freak colleagues. I’m afraid that they’re just not going to like giving up control like this.”

Chapel’s response: Multiresolution Language Design
Multiresolution Language Design: Motivation

Low-Level Machine-Oriented Concepts
- MPI
- OpenMP
- Pthreads

Target Machine

High-Level Abstractions
- HPF
- ZPL

Target Machine

“Why is everything so tedious?”
“Why don’t my programs port trivially?”

“Why don’t I have more control?”
**Multiresolution Design**: Support multiple tiers of features
- higher levels for programmability, productivity
- lower levels for performance, control
- build the higher-level concepts in terms of the lower

*Chapel language concepts*

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target Machine

- separate concerns appropriately for clean design
- yet permit the user to intermix the layers arbitrarily
Sample Lower-Level Chapel Concepts

Locality feature:

```chapel
on x do foo();
```

// run foo() on the locale storing x

Task parallel feature:

```chapel
coforall i in 0..#numTasks {
    foo(i);
}
```

// create a task per iteration

// join tasks before going on

Base language features:

```chapel
iter fib(n) {
    var current = 0, next = 1;
    for 1..n {
        yield current;
        current += next;
        next <=> current;
    }
}
```

// define an iterator

// use type inference

// generate next result

// swap operator

```chapel
for f in fib(10) do writeln(f);
```

// invoke iterator serially
A language can support both global- and local-view programming

```c
proc main() {
    coforall loc in Locales do
        on loc do
            MySPMDProgram(loc.id, Locales.numElements);
    }

proc MySPMDProgram(me, numCopies) {
    ...
}
```
A language can support both global- and local-view programming (and even message passing)

```c
proc main() {
    coforall loc in Locales do
        on loc do
            MySPMDProgram(loc.id, Locales.numElements);
    }

proc MySPMDProgram(me, numCopies) {
    MPI_Reduce(mySumOfSquares, sumOfSquares,
        MPI_SUM, MPI_DOUBLE, 0,
        MPI_COMM_WORLD);
}
```
Other Ways to Provide Additional Control

• **Interoperability**: support calls out to more traditional languages

```c
extern proc iMustWriteThisInC(x, y);
```

• also helps with adoption, preserving legacy code

• **Inlining Languages**: embed traditional languages within the higher-level language

```c
inline {
    ...(insert C code here)...
}
```

• like inlining assembly within C
Global-View: How Implemented in Chapel?

```chapel
config const n = computeProblemSize();
const D = [1..n];
var A, B: [D] real;

const sumOfSquares = + reduce (A**2 + B**2);
```

**Chapel:** Defined in terms of *zippered iteration* semantics
config const n = computeProblemSize();
const D = [1..n];
var A, B: [D] real;

const sumOfSquares = + reduce forall (a, b) in (A, B) do (a**2 + b**2);

Chapel: Defined in terms of zippered iteration semantics
Chained whole-array operations are implemented element-wise rather than operator-wise.

⇒ No temporary arrays required by semantics

\[
\begin{align*}
A**2 + B**2 & \Rightarrow T1 = A**2; \\
& \quad T2 = B**2; \\
& \quad T3 = T1 + T2;
\end{align*}
\]

⇒ forall \((a,b)\) in \((A,B)\) do \((a**2 + b**2)\);

• Provides an execution model that one can reason about and control using domain maps.
Domain Maps

**Domain Maps:** “recipes for parallel/distributed arrays and domains (index sets)”

Domain maps define:
- Mapping of domain indices and array elements to locales
- Layout of arrays and index sets in memory
- Standard operations on domains and arrays
  - e.g., random access, iteration, slicing, reindexing, rank change

Domain maps are built using Chapel concepts
- classes, iterators, type inference, generic types
- task parallelism
- locales and on-clauses
- other domains and arrays
Layouts and Distributions

Domain Maps fall into two major categories:

**layouts:** target a single locale (memory)
- e.g., a desktop machine or multicore node
- **examples:** row- and column-major order, tilings, compressed sparse row, space-filling curves

**distributions:** target distinct locales (memories)
- e.g., a distributed memory cluster or supercomputer
- **examples:** Block, Cyclic, Block-Cyclic, Recursive Bisection, ...
config const n = computeProblemSize();
const D = [1..n];
var A, B: [D] real;

No domain map ⇒ use default layout

const sumOfSquares = + reduce forall (a, b) in (A, B) do (a**2 + b**2);

Since A is first array in zippering, it is the leader.
Leader/Follower Iterators

- All zippered forall loops are defined in terms of leader/follower iterators:
  - *leader iterators*: specify parallelism, assign iterations to tasks
  - *follower iterators*: serially execute work generated by leader

- conceptually, the Chapel compiler translates:

  ```chapel
  forall (a,b) in (A,B) do
  (a**2 + b**2);
  ```

  into:

  ```chapel
  for work in A.lead() do
  for (a,b) in (A.follow(work), B.follow(work)) do
  yield a**2 + b**2;
  ```

  ...where A’s/B’s domain maps define *lead()* & *follow()*.
Defining Leaders and Followers

Leader iterators are defined using task/locality features:

```plaintext
iter BlockArr.lead() { 
    coforall loc in Locales do 
    on loc do 
        coforall tid in here.numCores do 
            yield computeMyBlock(loc.id, tid);
    }
```

Follower iterators simply use serial features:

```plaintext
iter BlockArr.follow(work) { 
    for i in work do 
        yield accessElement(i);
    }
```
config const n = computeProblemSize();
const D = [1..n];
var A, B: [D] real;

The default layout:
• targets local memory and processors only
• its leader iterator...
  ...by default, uses #tasks = #cores
  ...decomposes indices/elements using static blocking
Q: “So what do I... (oops, I mean) ... what does my control freak colleague do if s/he doesn’t like the leader iterator’s approach?”

A: Several possibilities...
config const n = computeProblemSize();
const D = [1..n];
var A, B: [D] real;

const sumOfSquares = + reduce forall (b,a) in (B,A) do (a**2 + b**2);

Make something else the leader.
(moot in this case – B also uses default domain map)
config const n = computeProblemSize();
const D = [1..n];
var A, B: [D] real;

const sumOfSquares = + reduce forall (a,b)
in (myLdr(A,blk=64), B)
do (a**2 + b**2);

Invoke some other leader iterator explicitly (perhaps one that you wrote yourself).
Config const n = computeProblemSize();
const D = [1..n] dmapped BlockCyclic(start=1, blocksize=64);

var A, B: [D] real;

const sumOfSquares = + reduce forall (a,b) in (A,B) do (a**2 + b**2);

Change the array’s default leader by changing its domain map (perhaps to one that you wrote yourself).
Controlling Data Parallelism: Hmm...

• We can still control an array’s decomposition, layout
• We can still control parallelism and work mapping
  • even explicitly if we want to (SPMD-in-Chapel)

**Maybe we can continue to be control freaks after all!**

(and really, did you think HPCers could be anything else?)

• Yet, by using domain maps & iterators, we...
  • insulate our algorithm from its implementation details
  • make the code more portable, readable, maintainable, etc.
  • and really, isn’t that what **productivity** is all about?)
For More Information on Domain Maps

- HotPAR’10 paper: *User-Defined Distributions and Layouts in Chapel: Philosophy and Framework*
- Upcoming CUG’11 paper/talk: *Authoring User-Defined Domain Maps in Chapel*

For Chapel users...

- Technical notes detailing domain map interface for programmers:
  
  $CHPL_HOME/doc/technotes/README.dsi

- Current domain maps:
  
  $CHPL_HOME/modules/dists/*.chpl
  
  layouts/*.chpl
  
  internal/Default*.chpl
Hardware wishlist for exploiting asynchrony:

- atomic operations (local and remote)
- single-sided puts/gets
- pervasive full/empty bits
- network support for active messages

Tools wishlist for asynchrony at exascale:

- relative/comparative debugging
- more pervasive visualization capabilities
  - of user data
  - of resource utilization: memory and processors
  - of algorithm execution
Takeaways: Asynchrony

*Data-oriented programming models help science to be insulated from implementation*
- yet, without necessarily abandoning control
- supports 90/10 rule well

*Building data parallelism using task parallelism supports asynchronous styles of parallelism*
- Results in execution models that are more general, dynamic, and loosely-coupled than today’s
- Serves as a good foundation for exascale
- Multiresolution philosophy is key here
Chapel ≠ HPF, due to its:

- well-defined execution model for data parallelism
- user-defined distributions & layouts
- ability to escape implicit data parallelism model

Chapel is not revisiting the HPF compilation problem

- rather, the “user-level specification of distributed parallel arrays” problem
For More Information

- **Chapel Home Page** (papers, presentations, tutorials): http://chapel.cray.com
- **Chapel Project Page** (releases, source, mailing lists): http://sourceforge.net/projects/chapel/
- **General Questions/Info:** chapel_info@cray.com (or chapel-users mailing list)

- **Upcoming workshop on this topic:**
  Future Approaches to Data-Centric Programming for Exascale (@IPDPS, May 20th 2011)