Parallel Computing: ways and means
Other options: TBB, c++11, cilk, Chapel

November 16, 2015
Lecture 27
Quote of the Day

“Choose a job you love, and you will never have to work a day in your life.”

-- Confucius
551 BC – 479 BC
Before We Get Started

- Issues covered last time:
  - MPI point-to-point communication: non-blocking send/receive
  - MPI collective actions:
    - Synchronization (barriers, wait, etc)
    - Communication (gather, scatter, etc.)
    - Operations (reduce, scan, etc.)

- Today’s topics
  - CUDA, OpenMP, MPI: putting things in perspective
  - Other parallel programming models, quick overview
    - TBB
    - C++11
    - Cilk
    - Chapel
    - Charm++ (next lecture)

- Other issues:
  - Wednesday is the last lecture of the semester
  - HW09, due on Wd, Nov. 18, at 11:59 PM
  - Final Project Proposal: if you don’t hear from me by Friday, it means that your proposal was fine
  - Second (and last) exam: coming up on 11/23 (Monday) at 7:15 PM (Room: 1610EH)
    - Review on 11/23 in 1610EH during regular lecture hours
MPI – We Scratched the Surface

- In some MPI implementations there are more than 300 MPI functions
- Not all of them part of the MPI standard though, some vendor specific

- Recall the 20/80 rule: six calls is probably what you need to implement a decent MPI code...
  - MPI_Init, MPI_Comm_Size, MPI_Comm_Rank, MPI_Send, MPI_Recv, MPI_Finalize
The PETSc Library
[The message: Use libraries if available]

- **PETSc: Portable, Extensible Toolkit for Scientific Computation**
  - One of the most successful libraries built on top of MPI
  - Intended for use in large-scale application projects,
  - Developed at Argonne National Lab (Barry Smith, technical lead)

- Provides routines for the parallel solution of systems of equations that arise from the discretization of PDEs
  - Linear systems
  - Nonlinear systems
  - Time evolution (numerical integration)

- Also provides utility routines, such as
  - Sparse matrix assembly
  - Distributed arrays
  - General scatter/gather (e.g., for unstructured grids)
Structure of PETSc

- Computation and Communication Kernels
  - MPI, MPI-IO, BLAS, LAPACK
- Profiling Interface
- PETSc PDE Numerical Solution Utilities
  - ODE Integrators
  - Visualization
  - Nonlinear Solvers, Unconstrained Minimization
  - Linear Solvers
  - Preconditioners + Krylov Methods
  - Object-Oriented Matrices, Vectors, Indices
  - Grid Management
  - Profiling Interface
- Visualization Interface
- Computation and Communication Kernels
  - MPI, MPI-IO, BLAS, LAPACK
## PETSc Numerical Components

<table>
<thead>
<tr>
<th>Nonlinear Solvers</th>
<th>Time Steppers</th>
<th>Krylov Subspace Methods</th>
<th>Preconditioners</th>
<th>Matrices</th>
<th>Distributed Arrays</th>
<th>Index Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Newton-based Methods</td>
<td>Other</td>
<td>Euler</td>
<td>Backward Euler</td>
<td>Pseudo Time Stepping</td>
<td>Other</td>
<td></td>
</tr>
<tr>
<td>Line Search</td>
<td>Trust Region</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Euler</td>
<td>Backward Euler</td>
<td>Pseudo Time Stepping</td>
<td>Other</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GMRES</td>
<td>CG</td>
<td>CGS</td>
<td>Bi-CG-STAB</td>
<td>TFQMR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Additive Schwartz</td>
<td>Block Jacobi</td>
<td>Jacobi</td>
<td>ILU</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compressed Sparse Row (AIJ)</td>
<td>Blocked Compressed Sparse Row (BAIJ)</td>
<td>Block Diagonal (BDIAG)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Distributed Arrays</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Flow Control for PDE Solution

Main Routine

Timestepping Solvers (TS)

Nonlinear Solvers (SNES)

Linear Solvers (SLES)

PETSc

PC

KSP

Application Initialization

Function Evaluation

Jacobian Evaluation

Post-Processing

User code

PETSc code
CUDA, OpenMP, MPI:
Putting Things in Perspective
Pros, CUDA

- Many remarkable success stories when the application targeted is data parallel and has high arithmetic intensity
  - Can provide sometimes one order of magnitude speed-ups

- Very affordable – democratization of parallel computing
  - At a price of $10K you get half the flop rate of what an IBM BlueGene/L delivered six or seven years ago

- Ubiquitous
  - Present on more than 100 million computers today support CUDA

- Good productivity tools
Cons, CUDA

- To extract last ounce of performance that makes GPU computing great you need to understand the computational model and the underlying hardware.

- Somewhat limited amount of device memory
  - 12 GB on K40 or 24 GB on K80

- Until the CPU and GPU are fully integrated, the PCI connection is impacting performance
  - To be addressed by NVLink next year: higher bandwidth, lower latency

- For true HPC, using CUDA in conjunction with MPI not a stroll in the park.
What Would Be Nice…

- CPU and GPU operate in the same physical memory space – essentially no penalty for main memory access from the GPU

- Get 3D memory out fast

- NOTE: what I had here two years ago got implemented
What Would Be Nice…
[the 2013 slide, two years ago]

- The global memory bandwidth should increase at least as fast as the rate at which the number of scalar processors increases.

- Integrate CPU & GPU so that concept of global device memory disappears.

- Have the OpenACC standard succeed for seamless parallel accelerator and/or many-core programming.
Pros of OpenMP

- Because it takes advantage of shared memory, the programmer does not need to worry (that much) about data placement.
- Programming model is “serial-like”, thus conceptually simpler than message passing.
- Compiler directives are generally simple and straightforward to use.
- Legacy serial code does not need to be rewritten.
Cons of OpenMP

- The model doesn’t scale up all that well

- In general, only moderate speedups can be achieved
  - Because OpenMP codes tend to have serial-only portions, Amdahl’s Law prohibits substantial speedups

- Amdahl’s Law:
  \[ s = \text{Fraction of serial execution time that cannot be parallelized} \]
  \[ N = \text{Number of processors} \]

\[
\text{Execution speedup: } \quad s + \frac{1 - s}{N}
\]

- Ideal targets for OpenMP: big loops that dominate execution time
Why Plot on Previous Slide Doesn’t Apply to MPI?

- Source of problem is different
  - MPI – multiple programs running on different nodes
    - The show stopper is the communication latency over interconnect
    - Each program most often sequential yet it might choose to use OpenMP
      - For the latter, the plot becomes relevant
  - OpenMP – running on one node
    - The show stoppers:
      - How much code you get to run under OpenMP (fraction of parallelism)
      - Cache coherence issues
Pros of MPI

- Good vendor support for the standard
  - It was great that the community converged upon a standard (something that can’t be said about GPU computing)

- Proven parallel computing solution, demonstrated to scale up to hundreds of thousands of cores

- Can be deployed both for distributed as well as shared memory architectures

- Today it is synonym with High Performance Computing
  - Provided a clear and relatively straightforward framework for reaching Petaflops grade computing
Cons of MPI

- Very targeted towards solving a certain type of computing problem
  - Very “Scientific Computing” oriented, particularly serving the domain decomposition folks (solution of PDEs, finite element)
    - Doesn’t handle heterogeneity all that well

- The interconnect is Achilles’ heel. Top bandwidths today are comparable to what you get over PCI-Express
  - Latency typically is significantly worse though

- Like CUDA, works well only for applications where you don’t have to communicate all that much (high arithmetic intensity)
HPC and MPI related blog post

- Please make sure you read this post:
  http://www.dursi.ca/hpc-is-dying-and-mpi-is-killing-it/
General Remarks on Parallel Computing

- Parallel Computing is and will be relevant at least for this decade

- Nonetheless, it continues to be challenging
  - Switching your thinking about getting a job done from sequential to parallel mode takes some time but it’s a skill that must be acquired
    - Parallel Programming more difficult than programming for Sequential Computing
  - Productivity tools (debuggers, profilers, build solutions) more challenging to master
  - Need to understand the problem that you solve, the pros/cons of the parallel programming models available, and of the hardware on which your code will run
Other Parallel Programming Alternatives

- Brought to you w/ input from Hammad and Tim Haines
Goal of This Segment

- “good to know” material, might provide some feature that simplifies your life

- No time for in-depth coverage

- Most often very similar in functionality provided to something that we have already covered
  - Chapel is somewhat different
    - Quite different than MPI, more akin to Charm++
    - We’ll discuss Charm++ on Wd
Intel® Threading Building Blocks (TBB)

- C++ library that abstracts loop-based, shared-memory parallelism like OpenMP into dynamic, composable parallelism
- Supports data-parallel and task-parallel programming paradigms
- You can explicitly control the threads, yet you don't have to
  - TBB Mantra: “Think tasks, not threads”
- Runtime system keeps track of threads and uses **work stealing** to keep each thread busy
  - Similar to OpenMP “tasks”

- Some features are drop-in replacements for standard C++ algorithms

```cpp
#include <tbb/parallel_sort.h>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> x(1000000);
    // fill vector (maybe from file, etc.)

    // Standard C++ sort is serial
    std::sort(x.begin(),x.end());

    // TBB sort is parallel
    tbb::parallel_sort(x.begin(),x.end());
}
```

[Tim Haines]→
#include <tbb/parallel_for.h>
#include <tbb/task_scheduler_init.h>
#include <vector>
#include <iostream>
#include <algorithm>
#include <iterator>

int main()
{
    // Initialize the TBB runtime system with two threads
    // NOTE: This is optional!
    tbb::task_scheduler_init init(2);

    constexpr size_t N = 100000;
    std::vector<float> x(N);
    using size_type = decltype(x)::size_type;
    using value_type = decltype(x)::value_type;

    // process grain_size many elements per task.
    constexpr size_type grain_size = 10000;

    /* We have 10 tasks (N/grain_size), but only two threads!
    * The runtime system shuffles work between the threads
    * using work stealing.
    */
    tbb::parallel_for(size_type(0), N, grain_size,
                      [&x,grain_size](size_type i) {
                          auto start = x.begin() + i;
                          std::transform(start, start+grain_size, start,
                                          [
                                              x
                                          ] (const value_type &v) { return 2+v; });
                      },
                      []()
                      { return true; });

    // print the first few (should all be just 2)
    std::copy(x.begin(), x.begin() + 10,
              std::ostream_iterator<value_type>(std::cout, " "));
    std::cout << std::endl;
}

The TBB sort ~3x faster on Tim’s desktop
Intel® Threading Building Blocks (TBB)

- **TBB Pros:**
  - Simple algorithm-centric library written in portable C++11
  - Dynamic, data- and task-based parallelism
  - Provides several concurrent (lock-free) data structures
  - Can easily be incrementally introduced into existing sequential code
  - Offers an aligned, cache-aware memory allocator
  - Plays nice with MPI, processor affinity, and ccNUMA
Intel® Threading Building Blocks (TBB)

- **TBB Cons:**
  - Target machine must have entire library installed (including platform-specific memory allocator)
  - Doesn't always play nice with other shared-memory libraries like OpenMP, pthreads or C++ std::thread
Intel® Threading Building Blocks (TBB)

- TBB opens multiple opportunities for parallelism
  - parallel_do
  - pipeline
  - filter
  - recursive splitters
  - etc.

- Reference:
C++11 Thread Support Library

- Native, task-based parallelism

- Sophistication level:
  - Between that of pthreads and a “high-level” threading library like TBB

- Bottom line: a collection of type-safe threading features upon which larger libraries can be built in a platform-independent fashion

- It enables more flexible task-based parallelism than OpenMP's 'task' or 'sections'
  - In particular, it actually handles exceptions

```cpp
#include <vector>
#include <algorithm>
#include <iostream>
#include <numeric>
#include <stdexcept>
#include <future>
#include <iterator>
#include <type_traits>

class empty_container: public std::exception {
  constexpr static const char* msg = "Empty container";
public:
  virtual const char* what() const noexcept override {
    return msg;
  }
};

int main() {
  auto mean = [](const auto begin, const auto end) { // C++14 polymorphic lambda
    if(auto size = std::distance(begin,end)) {
      return std::accumulate(begin,end,
                               typename std::decay<decltype(*begin)>::type()) / size;
    }
    throw empty_container();
  };

  auto inclusive_scan = [](const auto begin, const auto end) {
    if(auto size = std::distance(begin,end)) {
      std::vector<typename std::decay<decltype(*begin)>::type> sum(size);
      std::partial_sum(begin,end,sum.begin());
      return sum;
    }
    throw empty_container();
  };

  std::vector<int> x(10000);
  std::iota(x.begin(), x.end(), 1);
  // These calls do not block
  auto avg = std::async(std::launch::async, mean, x.begin(), x.end());
  auto inc_scan = std::async(std::launch::async, inclusive_scan, x.begin(), x.end());

  // Calling get() blocks the future until the respective computation is done
  std::cout << "average = " << avg.get() << std::endl;

  auto is = inc_scan.get();
  std::copy(is.begin(), is.begin() + 10, std::ostream_iterator<decltype(x)::value_type>(std::cout, " "));
  std::cout << std::endl;
}
```
Intel Cilk™ Plus

- Extends C/C++ with a handful of keywords
- Every Cilk program has serial semantics
- Cilk provides performance guarantees based on performance abstractions
- Cilk is processor-oblivious
- Cilk’s runtime system automatically manages low-level aspects of parallel execution, including protocols, load balancing, and scheduling
- Cilk supports speculative parallelism
Intel Cilk™ Plus

- Cilk Plus defines three keywords
  - **cilk_spawn**
    - Allows runtime to spawn thread if needed
    - Will work steal from other cores if core doesn’t have enough work
  - **cilk_sync**
    - Wait for spawned child functions to finish
  - **cilk_for**
    - Optimized for loop construct
Fibonacci Example

```c
int fib (int n) {
    if (n<2) return (n);
    else {
        int x,y;
        x = fib(n-1);
        y = fib(n-2);
        return (x+y);
    }
}
```

Cilk code

```c
int fib (int n) {
    if (n<2) return (n);
    else {
        int x,y;
        x = cilk_spawn fib(n-1);
        y = cilk_spawn fib(n-2);
        cilk_sync;
        return (x+y);
    }
}
```

- Cilk is a faithful extension of C
- A Cilk program’s “serial elision” is a valid C program
- Cilk provides no new data types
Basic Cilk Keywords

Cilk code

```c
int fib (int n) {
    if (n<2) return (n);
    else {
        int x,y;
        x = cilk_spawn fib(n-1);
        y = cilk_spawn fib(n-2);
        cilk_sync;
        return (x+y);
    }
}
```

The named **child** Cilk procedure can execute in parallel with the **parent** caller.

Control cannot pass this point until all spawned children have returned.

This is basically identical to OpenMP's "task" Fibonacci example.
Parallel For Loop

C code

```c
for (int i = 0; i < 8; ++i) { do_work(i); }
```

Cilk code (not optimal)

```c
for (int i = 0; i < 8; ++i) {
    cilk_spawn do_work(i);
}
cilk_sync;
```

Optimal cilk code

```c
cilk_for (int i = 0; i < 8; ++i
    { do_work(i); }
```

- Similar to spawning OpenMP tasks
- Similar to OpenMP “parallel for”
Difference Between spawn and for

```c
for (int i = 0; i < 8; ++i)
    {cilk_spawn do_work(i); } cilk_sync;
```

cilk_for (int i=0;i<8;++i){ do_work(i);}

[Hammad]→
Intel Cilk Plus
Reducer Library

- Reducers are lock free structures for parallel applications
- Example: Fibonacci with an add reducer

```c++
cilk::reducer< cilk::op_add<int> > fib_sum(0);

void fib_with_reducer_internal(int n) {
    if (n < 2) {
        *fib_sum += n;
    }
    else {
        cilk_spawn fib_with_reducer_internal(n-1);
        fib_with_reducer_internal(n-2);
        cilk_sync;
    }
}
```
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reducer_list_append</td>
<td>Creates a list by adding elements to the back.</td>
</tr>
<tr>
<td>reducer_list_prepend</td>
<td>Creates a list by adding elements to the front.</td>
</tr>
<tr>
<td>reducer_max</td>
<td>Calc max value of a set of values.</td>
</tr>
<tr>
<td>reducer_max_index</td>
<td>Calc max value and index of that value of a set of values.</td>
</tr>
<tr>
<td>reducer_min</td>
<td>Calc min value of a set of values.</td>
</tr>
<tr>
<td>reducer_min_index</td>
<td>Calc min value and index of that value of a set of values.</td>
</tr>
<tr>
<td>reducer_opadd</td>
<td>Calc sum of a set of values.</td>
</tr>
<tr>
<td>reducer_opand</td>
<td>Calc binary AND of a set of values.</td>
</tr>
<tr>
<td>reducer_opor</td>
<td>Calc binary OR of a set of values.</td>
</tr>
<tr>
<td>reducer_opxor</td>
<td>Calc binary XOR of a set of values.</td>
</tr>
<tr>
<td>reducer_string</td>
<td>Accumulates a string using append operations.</td>
</tr>
<tr>
<td>reducer_wstring</td>
<td>Accumulates a &quot;wide&quot; string using append operations.</td>
</tr>
<tr>
<td>reducer_ostream</td>
<td>An output stream that can be written in parallel.</td>
</tr>
</tbody>
</table>
The Intel Cilk™ Plus Development Tools

- **Cilk Screen**
  - Reports race conditions in Cilk Plus constructs that could be encountered during execution

- **Cilk View**
  - Helps w/ gauging of the performance of a Cilk Plus parallel program
  - Predicts how performance will scale on multiple processor systems
  - Automatically benchmarks a Cilk Plus program running on one or more processors
Cilk Plus Array Notation

- Extension to C++ for parallel operations on arrays

\[
A[start\_index:length:stride] = \ldots
\]

- Static array operations can be simplified

```
for (i = 0; i < 10; i++)
D[i] = A[i] + B[i];

D[:] = A[:] + B[:];
```

- Dynamic arrays need a start and length

\[
A[start\_index:length] = \ldots
\]
Cilk Plus Array Notation

- Scatter and gather operations are simplified

\[ C[::] = A[B[::]] \]
  - Gather elements to C from A given indices in B

\[ A[B[::]] = C[::] \]
  - Scatter elements to A given indices in B from C
Cilk Plus
Built-in Functions for Array Sections

- Build in reduce functions for arrays

  __sec_reduce_add (A[:])
  - Returns sum of elements

  __sec_reduce_max (A[:])
  - Returns max element

  __sec_reduce_max_index (A[:])
  - Returns index of max element
**SIMD enabled functions**

- Adding `__declspec(vector)` to a function will generate vectorized code

```c
__declspec(vector) float myfunction(float s, float k, float r);

void Fun(float S[SIZE], float r, float out[SIZE]){
    out[:] = myfunction(S[:], r);
}
```

`SIZE` is defined elsewhere.
The vectorization magic happens at compile time.
Force Loop Vectorization

- Use `#pragma simd` to enforce loop vectorization
- Compile will give warning if code cannot be vectorized

```c
void add_floats(float *a, float *b, float *c, float *d, float *e, int n)
{
    int i;
    #pragma simd
    for(i=0;i<n;i++){
        a[i]=a[i]+b[i]+c[i]+d[i]+e[i];
    }
}
```
Chapel

- **Chapel: Cascade High-Productivity Language**
  - Promoted by Cray (since 2009)

- **Overall goal:** “solve the parallel programming problem”
  - Simplify the creation of parallel programs
  - Support their evolution to extreme-performance, production-grade codes
  - Emphasize generality

- **Desirable attributes, guiding the language design process:**
  1) Multithreaded parallel programming
  2) Locality-aware programming
  3) Object-oriented programming
  4) Generic programming and type inference
A Simple Domain Declaration

var m: integer = 4;
var n: integer = 8;

var D: domain(2) = [1..m, 1..n];
A Simple Domain Declaration

\begin{verbatim}
var m: integer = 4;
var n: integer = 8;

var D: domain(2) = [1..m, 1..n];
var DInner: domain(D) = [2..m-1, 2..n-1];
\end{verbatim}
Domain Uses

- Declaring arrays:
  ```
  var A, B: [D] float;
  ```

- Sub-array references:
  ```
  A(DInner) = B(DInner);
  ```

- Sequential iteration:
  ```
  for (i, j) in DInner { ...A(i, j)... }
  or: for ij in DInner { ...A(ij)... }
  ```

- Parallel iteration:
  ```
  forall ij in DInner { ...A(ij)... }
  or: [ij in DInner] ...A(ij)...
  ```

- Array reallocation:
  ```
  D = [1..2*m, 1..2*n];
  ```
Other Arithmetic Domains

\begingroup
\small
\textbf{var} \ D2: \ \textit{domain}(2) = (1,1)..(m,n);

\textbf{var} \ StridedD: \ \textit{domain}(D) = D \ by \ (2,3);

\textbf{var} \ indexList: \ \textit{seq}(\textit{index}(D)) = \ldots;
\textbf{var} \ SparseD: \ \textit{sparse} \ \textit{domain}(D) = \textit{indexList};
\endgroup

- These are language facilities to map computations to domains
Task Parallelism

- **co-begins**: indicate statements that may run in parallel:

  ```
  computePivot(lo, hi, data);
  cobegin {
    cobegin {
      ComputeTaskA(...);
      Quicksort(lo, pivot, data);
      Quicksort(pivot, hi, data);
    }
  }
  ```

- **atomic sections**: support atomic transactions

  ```
  atomic {
    newnode.next = insertpt;
    newnode.prev = insertpt.prev;
    insertpt.prev.next = newnode;
    insertpt.prev = newnode;
  }
  ```

- **sync and single-assignment variables**: synchronize tasks

  [Hammad]→
Locality-aware Programming

- **locale**: machine unit of storage and processing

- Programmer specifies number of locales on executable command-line
  
  ```
  prompt> myChapelProg -nl=8
  ```

- Chapel programs provided with built-in locale array:
  
  ```
  const Locales: [1..numLocales] locale;
  ```

- Users may use this to create their own locale arrays:
  
  ```
  var CompGrid: [1..GridRows, 1..GridCols] locale = ...;
  ```

  ![CompGrid diagram]

  ```
  var TaskALocs: [1..numTaskALocs] locale = ...;
  var TaskBLocs: [1..numTaskBLocs] locale = ...;
  ```

  ![TaskALocs and TaskBLocs diagrams]
Data Distribution

- domains may be distributed across locales
  
  \[
  \text{var } D \text{: domain(2) distributed(Block(2) to CompGrid)} \quad = \ldots;
  \]

- Distributions specify...
  - mapping of indices to locales
  - per-locale storage layout of domain indices and array elements

- Distributions implemented as a class hierarchy
  - Chapel provides a number of standard distributions
  - Users may also write their own

[Hammad]→
Computation Distribution

- “on” keyword binds computation to locale(s):

```plaintext
cobegin {
  on TaskALocs do ComputeTaskA(...);
  on TaskBLocs do ComputeTaskB(...);
}
```

- “on” can also be used in a data-driven manner:

```plaintext
forall (i,j) in D {
  on B(j/2, i*2) do A(i, j) = foo(B(j/2, i*2));
}
```
Other Chapel Features

- Tuple types, type unions, and typeselect statements
- Sequences, user-defined iterators
- Support for reductions and scans (parallel prefix) including user-defined operations
- Default arguments, name-based argument passing
- Function and operator overloading
- Modules (for namespace management)
- Interoperability with other languages
- Garbage Collection
Chapel Challenges

- User Acceptance
  - True of any new language
  - Skeptical audience

- Commodity Architecture Implementation
  - Chapel designed with idealized architecture in mind
  - Clusters are not ideal in many respects
  - Results in implementation and performance challenges

- Cascade Implementation
  - Efficient user-defined domain distributions
  - Type determination w/ OOP w/ overloading w/ …
  - Parallel Garbage Collection

- And many others as well…
Summary

- Chapel designed to
  - Enhance programmer productivity
  - Address a wide range of workflows

- Relies on high-level, extensible abstractions for
  - Global-view multithreaded parallel programming
  - Locality-aware programming
  - Object-oriented programming
  - Generic programming and type inference

- Status:
  - draft language specification available at: http://chapel.cs.washington.edu
  - Open source implementation proceeding apace