A Comparison of Unified Parallel C Titanium and Co-Array Fortran
(parallel computing made fun, easy and entertaining)
So you want a parallel language, do you?

- Compiler Extensions
  - Like OpenMP
- Entirely New Languages
- Language Extensions
  - UPC, Titanium and Co-Array Fortran
What to add?

• A means of parallelism!
  – Multiple processes or threads
  – Some means of work sharing
• A way to create global data
  – Simple and easy is nice
  – Complex and messy, not nice
• Synchronization
Goal of this Project

• Originally
  – Wanted to compare the ways a parallel task was represented
  – Expected some elaborate and different way of automatically dividing out work, like a better version of OpenMP
  – Found everything was more dependent on the representation of the data
Goal Continued

• Revised Plan
  – To compare the languages in terms of how the representation of the data affects the means of parallelization.
  – Figure out why Fortran has (*’s at the end of its arrays! (No bounds checking, evidently, but that’s something for later, or, perhaps never!)
Onward to the comparing!
Unified Parallel C

"If you were plowing a field, what would you rather use? Two strong oxen or 1024 chickens?"
-Seymour Cray
Unified Parallel C -- Overview

• Same old C, fun new features
• Shared arrays, shared pointers and shared pointers to shared arrays
• An assortment of MPI-lish barriers and fences.
• Explicit Parallelism!
  – upc_forall
UPC -- Overview

• Logically modeled as a bunch of threads in a shared address space
• SPMD
• The threads are actually processes and can exist locally or remotely
• Communication is handled by your choice of a bunch of options (MPI, ARMCI, sockets)
UPC – Shared Memory

• The “shared” keyword
  – shared int goat[THREADS];
  – shared double donkey;
  – shared [10] double weasel[THREADS][10];
  – shared [20] int lemur[20][40];

• So, what will this do?
  – Thread #2 accesses lemur[20][5]
UPC – upc_forall

- Nice feature of UPC
- Similar to OpenMP’s parallel for
- \textit{upc\_forall} (\textit{init}; \textit{test}; \textit{loop}; \textit{affinity})
  - Init, test and loop are the same as normal C
  - The affinity statement allows some cool stuff
  - Can be either:
    - Continue – not too interesting
    - Pointer to shared
    - Integer expression
“If you have a million monkeys and a million typewriters, how long until one of them codes homework #5 for me?”

“I don’t know, but not by Friday.”

“Looks like I need more monkeys…”
Titanium – High Performance Java

• Java?

• Uses java syntax as a base
  – Perhaps a new language rather than a language extension

• Discards all the “java stuff”
  – No JVM
  – “Immutable Classes” – Objects that are stored directly, rather than by pointers
Titanium -- Overview

- No JVM? Direct stack-based storage? Sounds suspiciously like C!
- Titanium is a two part compiler
  - Titanium itself takes Java code and turns it into C code
  - A backend compiler (your choice) takes the C code and makes your executable
- The Titanium compiler itself is written in C++
Titanium -- Overview

• SPMD model, like UPC
• Threads in a shared address space, like UPC
  – If you have a reference to something, you can use it
  – You don’t necessarily have all the references though!
• Must explicitly communicate with other threads to get references to shared data
Titanium – Region-Based Memory

• No garbage collection
• Allocate (via new) within a region
• When the region is no longer needed, destroy the region
• Cleans up all data structures contained within the region (designed to avoid collection problems with circular lists)
Titanium – Regions, Domain and Points

- No java arrays
- Variably Sized *Domains*
  - RectDomain
    - Determined by 2 *Points<dim>*, the upper left and lower right
    - Arrays can be allocated based on these domains
  - Domain
    - Union of RectDomains, allowing for variably sized rows/columns (or other, non-matrix, shapes!)
Titanium – Domains, Points, Arrays

Generating a 20x20 matrix:

```java
Point<2> upper_left = [1,1];
Point<2> lower_right = [20,20];
RectDomain<2> r = [upper_left : lower_right];
double [2d] A = new double[r];
```
Titanium – Unordered Iteration

• The *foreach* (*<point>* in *<domain*>) statement allows unordered iteration
• Compiler can reorder communication for efficiency, based on locality
• Supposedly does a good job due the limited nature of the language (fewer things to screw up optimization)
Titanium – Unordered Iteration

• Here’s an example (accumulating all elements in our earlier matrix, in no particular order)

```java
double acc;
foreach (p in A.domain())
{   acc += A[p];
}
```
Titanium -- foreach

- *foreach* is **NOT** parallel!
- If your domain is the entire set of data, every thread will work on every piece of the data!
- Oh no!
- Divide out your regions appropriately, then make sure the references to regions get to where they should be.
Co-Array Fortran

“Fortran’s not a dead language. It’s an undead language!”
Co-Array Fortran -- Overview

• Co-Array Fortran, like the others, is a language extension (of Fortran 95 – not Fortran 77!)
• Major Feature: Co-Arrays!
• Also an SPMD language
• Like UPC, it adds some valuable features, while leaving the rest of the language mostly the same.
Co-Array Fortran – Overview

- No explicit structures for parallelism!
- Depends on image ID
  - Image number = process ID = thread number (in general)
- Must explicitly determine locality information (versus UPC’s pointer affinity)
Co-Array Fortran – Shared Memory

• The mighty power of the Co-Array!
• Normal arrays are turned into co-arrays by adding an extra set of dimensions after the normal array dimensions:

  real, dimensions(10)(10) -- (normal)
  real, dimensions(10)(10)[*] -- (co-array)
Co-Array Fortran – Memory Access

• Getting stuff out of the co-arrays works like UPC – specify an address, get the element.

\[ a(5)(5)[3] \quad \text{Retrieves element} \quad a(5,5) \text{ from image 3} \]
Co-Array Fortran – Co-Arrays

• Allows more flexibility in data distribution than Titanium or UPC
• Operating on local data relies on MPI-like checks of image ID.

Begone, evil zombie language!
## Pretty Graphs and Pictures!

<table>
<thead>
<tr>
<th></th>
<th>Shared Memory</th>
<th>Explicit Parallelism</th>
<th>Synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPC</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Titanium</td>
<td>yes/no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Co-Array Fortran</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
Pretty Charts and Graphs!

<table>
<thead>
<tr>
<th></th>
<th>Single Image of Shared Data?</th>
<th>SPMD?</th>
<th>Way easier than all the others?</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPC</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Titanium</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Co-Array Fortran</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>
Conclusions

• They all work
• What do you gain by using one of these languages versus, say, C and MPI/GA?
• As development goes on, hopefully they become simpler
• UPC and Co-Array Fortran seem equivalent
• Titanium is more specialized
Areas for Improvement

• I’d hoped to find better ways of:
  – Representing shared data
    • Limited to arrays
    • What about more complex data structures?
      – Trees, graphs, etc
      – Is it possible?
  – Representing parallel tasks
    • This wasn’t answered
    • Different paradigm, perhaps?
    • Still a serial language for a parallel task
Areas for Improvement

• How about Meta-Languages?
  – An “in the middle” sort of thing, with generic methods of expressing shared data and shared tasks
  – Again, maybe possible, maybe not.

• How easy can the languages be to use?
  – UPC seems pretty easy.
  – Can the parallelism happen without the programmer knowing anything?