Shaun Lindsay

CS425

A Comparison of Unified Parallel C, Titanium and Co-Array Fortran

The purpose of this paper is to compare Unified Parallel C, Titanium and Co-Array Fortran's methods of parallelism and shared data handling. In particular, I will discuss Unified Parallel C and compare the other languages to it. This is due in part to a larger set of features present in Unified Parallel C and also to my greater degree of familiarity with UPC versus Titanium and Fortran in general.

Unified Parallel C

Unified Parallel C (referred to as UPC) is an extension to the C standard, developed over the last several years. Implementations have been built by three major universities, George Washington University, Michigan Tech and UC Berkeley.

UPC adds the constructs for the two usual purposes of distributed memory management and handling parallelism, but also adds the ability to explicitly specify parallelism (ala OpenMP). Like the other languages, UPC's handling of parallelism is closely bound to the way it deals with shared memory.

Shared Memory

In keeping with the C style of memory management, UPC is centered around the use of shared arrays, accessed either directly or via a pointer to a shared mechanism. These arrays may either be allocated globally on the stack, or allocated dynamically at runtime. Due to scoping problems, no auto declarations are allowed (i.e. no creation of shared arrays within functions).

Declaring shared arrays and shared pointers is fairly straightforward:

shared int x [THREADS];	Allocates an array of integers with each thread owning one element.
shared int y [10][THREADS];	Allocates a 10xTHREADS matrix with each thread owning a 1x10 column.
shared int a;	Allocates one copy of the globally accessible integer a. (a is owned by thread 0)
int b;	Allocates like normal – the integer b is private to the thread that declares it.

The keyword "shared" specifies that the memory being allocated is globally accessible, but distributed across the various threads in some fashion. Unless explicitly stated by the programmer, each thread will get an element in a round-robin fashion:

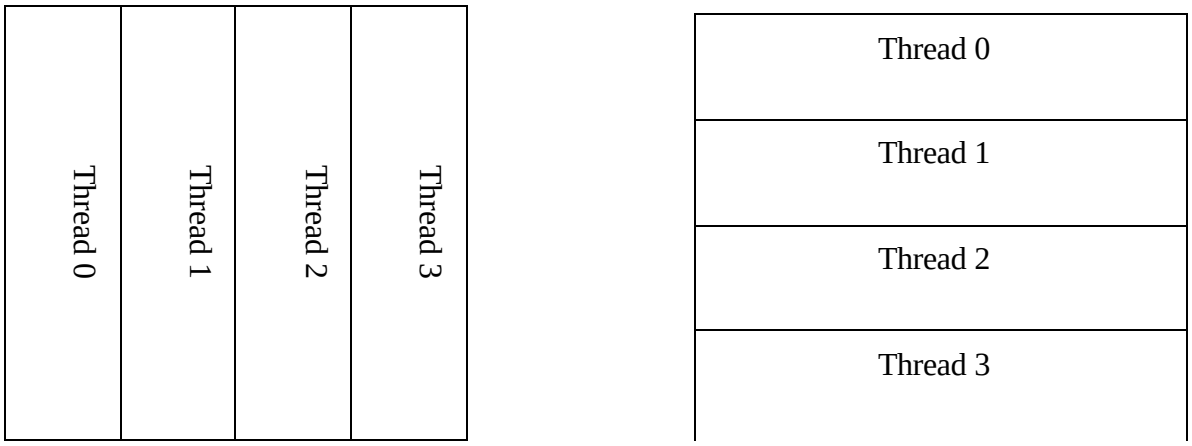
x[0] -> Thread 0, x[1] -> Thread 1, x[2] -> Thread 3, etc.

Automatic distribution is convenient, but some flexibility is necessary to control the allocation of data across the threads. This is done by specifying another index between the "shared" and the <type> in the statement as follows:

shared [THREADS] int a[THREADS][THREADS];

Let's assume for the above example that THREADS = 4. This statement is going to allocate a 4x4 matrix which is globally accessible, as it would without the distribution index, but instead of giving each thread a column, this will give each thread a row.

Memory Distributions for
shared int [THREADS][THREADS]
vs.
shared [THREADS] int [THREADS][THREADS]



Pointers

Since UPC tries to retain as much of the C style memory methods as possible, it has pointers to shared data structures, operating very much like standard C's pointers. The main difference is that there are three fields to the pointers: *thread number, local address and phase.* The *thread number* specifies which thread owns the element the pointer is currently pointing to. The *local address* is the location (in local memory) at which the current element resides. The *phase* is an integer that specifies where in the local block the pointer is pointing. If, like in the previous example, we specified a matrix with the statement:

shared [4] int [4][4];

then the phase of a pointer to that matrix can range from *0-3*. After the pointer is incremented past 3, it then points to the next thread and is reset to zero.

Affinity and upc_forall

UPC is unique amongst these languages because it has a method for explicitly handling parallelism (similar, in a sense, to OpenMP's *parallel for* directive). However, in keeping with the distributed nature of UPC, *upc_forall* allows work-sharing to be determined by data-locality, via an *affinity expression*.

The syntax for *upc_forall* is very similar to a standard C *for* loop, with one extra expression added at the end:

upc_forall (init; test; loop; affinity-expression) statement

The first three parameters are exactly as in normal *for* loop. The difference comes from the *affinity-expression* at the end. This determines which threads execute which iterations of the loop. There are three ways in which to do this:

1) affinity-expression = continue

2) affinity-expression = integer expression

3) affinity-expression = pointer-to-shared expression

The first option, *continue*, has all threads execute all iterations of the loops – thus, not accomplishing much in terms of parallelism. The second two divide out the work between threads in interesting ways:

If the *affinity-expression* is evaluated to an integer, this specifies the thread ID of the executing thread. So, to make every thread get a different iteration you could write:

upc_forall (i = 0; i < 20; i++; i % THREADS) {work();}

A more versatile way of using affinity is via the shared-pointer expressions. If a pointer to shared data is used, the thread owning the data pointed to gets that iteration of

the loop. If we wanted to increment every element of an array by one and have everything occur locally on the threads that own the data we could write:

```
upc_forall (i = 0; i < 20; i++; &p_array[i])
        {  p_array[i] += 1;  }
```

Assuming that p_array was an array of 20 elements and that it was distributed across several threads, this loop would have each thread increment all of the elements it owned. (Note that the address of a shared element is used – not the dereferenced pointer itself)

Synchronization

UPC has the expected assortment of synchronization constructs. All synchronization happens explicitly through one of the following functions:

upc_barrier	Blocks until all threads reach the barrier.
upc_wait	Blocks until a upc_notify is issued
upc_notify	Non-blocking call to restart a waiting thread
upc_fence	One-sided blocking call that waits until all shared memory references have resolved before continuing.
upc_lock **upc_lock_attempt** **upc_unlock**	These three functions allow shared data to be locked.

Consistency

Finally, UPC, like Titanium, has two modes for memory consistency. *Strict mode* allows no reordering of memory accesses. Everything will be done as the programmer

specifies. Alternatively, *relaxed mode* allows the compiler to reorder memory accesses to overlap communication and computation as it sees fit.

Titanium: High Performance Java

Titanium, like UPC, is a language extension, adding implicit parallelism to Java while keeping most of the syntax intact. Since Titanium was designed with no allegiance toward C syntax, nor to Java's memory handling, its methods of dealing with shared memory are very different.

Most importantly, Titanium is Java in name only. The designers took the syntax of the language as a base, then built an entirely different language from that. There's no JVM, they've added immutable classes (objects that can exist on the stack and be passed by value) and abandoned garbage collection in favor *region-based* memory management. In fact, the Titanium compiler actually compiles to C. The actual executable comes from a back-end C compiler of the user's choice (such as gcc).

The goals of the language seem to be different than those of UPC. UPC's flexibility and power come at the cost of being complicated and (relatively, at least) difficult to learn. Titanium is a simpler design, but due to that simplicity loses the expressiveness of UPC's *shared arrays* and *affinity expressions*.

Regions, Domains and Points

The major difference is the introduction of the aforementioned data structures. To avoid Java's dependence on garbage collection to deal with free memory, Titanium uses the concept of a *region*. *Regions* are areas of memory in which other data structures can be allocated. Everything inside a *region* can be explicitly destroyed by destroying the *region* itself.

So, the important concern then is "how do you create arrays inside regions?" Again, Titanium avoids using the Java arrays, which like everything else native to Java are objects the programmer has no control over. Inside, they've created *domains* of *points* which can then be changed into arrays. Part of the reason behind this is to allow the creation of non-rectangular domains (imagine a matrix, but each row is of arbitrary width).

The more specific, matrix-esque, structure is the *RectDomain*. This is used in the following fashion:

```
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];
```

These four lines of code allocate two points, then a rectangular domain with those two points as its corners. Finally, a two-dimensional array, A, is allocated over the *RectDomain*, creating the actual matrix, a two dimensional array, A[20,20]

Distributed – Shared Memory?

The concept of *regions* is an interesting work-around to get by Java's lack of explicit memory control, but it doesn't provide any faculty for global data structures. The memory model used for multiple processes in Titanium is that there is a global address space – if you get a reference to an object from another process, then you can use that reference as if it were local to your process. However, the catch is that the references must be passed via communication primitives, such as *ti.broadcast* and *ti.exchange*. So, unlike UPC, there isn't a coherent, overall feel to the arrays. Additionally, these are not

one-sided operations. Every process must participate, so synchronizing all the processes becomes more like MPI send and receives during the initialization. Afterward, however, all memory accesses are coded as if the memory was shared.

Unordered Iteration

All of the difficulty of setting up the *regions* and *domains*, in a sense, pays off with the introduction of unordered iteration. The Titanium *foreach* statement allows iteration over the points in a *domain*, in a non-deterministic fashion. For example, using the array from the earlier example:

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

This code creates a variable, acc, which, when the loop terminates, will contain the sum of all the elements in the array, A, done in no particular order.

An important concept with the *foreach* statement is that it does not imply parallelism. If all the threads execute a *foreach* statement over a give region, all the threads will do all the work, for no speed-up. Parallelism in Titanium is purely implicit, like Pthreads, where everything must be specified based on thread ID's. For the purposes of matrices, this would amount to each thread iterating over a seperate *domain*, which is up to the programmer to define. The advantage of the *foreach* statement, then, is that the compiler can order the operations to overlap as much of the communication as possible. Since the program is considered to be running in a global address space, the Titanium compiler is supposedly capable of doing all sorts of impressive static analysis, resulting in good speedup. All of the applications cited for speed increases didn't have accurately

corresponding counterparts written in other languages (mostly referenced were a partial differential equation solver and a heart simulator), so the speedup data is uncertain.

Co-Array FORTRAN

Co-Array FORTRAN, like UPC and Titanium, is a language extension. The major difference is the addition of co-arrays (as the name would suggest) and some primitives that allow the synchronization of multiples processes. There are no explicit structures for parallelism, instead handling the division of work via process image ID (like Titanium or Pthreads).

A co-array is analogous to UPC's shared array, but with a twist. The outermost indices of co-arrays indicate the locality of the data. This contrasts with UPC's uniform indexing method. Co-arrays essentially add another set of dimensions to an array that denote which piece of the array resides on which process. The distribution or the data is explicitly decided in the structure of the co-arrays, rather than by specifying sharing information (UPC) or multiple regions (Titanium).

Allocating co-arrays can be done statically or dynamically (but not automatically in subroutines, again for the same scoping issues as UPC). The co-dimensions are specified using square brackets containing the new outer dimensions, always placed after the round brackets which contain the local array sizes.

real, dimension(10,10)[*] :: a

Creates a co-array of local 10x10 matrices on an arbitrary number of images. Accessing this object is done by specifying the address in the same round/square bracket syntax.

For dynamically allocated arrays, FORTRAN's allocation syntax has been extended to account for square bracket convention.

real, allocatable :: a(:)[:]

allocate (array(10)[*])

Any allocation must occur on all processes before execution may continue.

Synchronization

Co-array FORTRAN has a similar set of synchronization constructs to UPC. These include *sync_all*, which acts as a barrier statement for all process images, *sync_team*, which works as a barrier between specific sets of processes, and *start_critcal / end_critical*, which allow for critical sections where only one process may be executing at a time.

Areas for Improvement

Of the three languages, only UPC has a method for explicit parallelism. Its *upc_forall* loop allows a high level of control of execution, something not present in Titanium and not (for a non-fortran programmer) as easy to achieve in Co-Array Fortran. It seems like a similar type of statement would be possible in Co-Array Fortran, using the co-array dimensions as the determiners for execution. Perhaps this is a flawed idea, however, since selecting execution based on image number in Co-Array Fortran doesn't seem to be that big of a problem, so the need might not be there.

All the languages seem to still be in early stages of development. The standards are there, but the refinements that come with use don't seem to be. For instance, the only data structures supported are multi-dimensional arrays. Granted, that's mostly all that's necessary for a lot of applications, but a more elaborate set of data structures (and support

for efficient distribution and apparent shared access) would be a useful addition. Possible additions would be things like trees or graphs, which could likely be modeled as arrays under the surface, but to the programmer, they would appear to be like any other structure.

In general, the goal of these languages is to make parallel computing easier, less error-prone and more efficient. While these languages hide the complexity of MPI-like interfaces, where communication has to be explicitly controlled, they are a long way from being simple, straightforward ways to implement parallel solutions. The best improvement in this area will probably be in the compiler's ability to analyze and exploit parallelism in algorithms written in these languages, allowing the programmer to specify less, but get more.

Another interesting possibility is to design a common language specifically designed towards expressing parallelism. If all the major ideas of these languages could be distilled down into a reasonable kernel, implementing new parallel languages based on that kernel would be a matter of constructing a different syntax for an existing implementation. This might not be possible, but if it is, a common standard would lessen the learning curve for parallel programming and make for more robust implementations.

Overall, UPC and Co-array Fortran seem to be computationally equivalent models. Anything that UPC can do, I imagine that co-array fortran could do using a slightly different implementation. The same, however, can not be said for Titanium. It seems to be a more specialized language, suited towards tasks with easily defined regions that are inherently data parallel where execution order is unimportant. The major

differentiating factor, in my opinion, is UPC's *upc_forall* statement. That alone makes it a more powerful and versatile language.