Improving Chapel and Array Memory Management

Michael Ferguson, Cray Inc (presenting author)
Vassily Litvinov, Cray Inc
Brad Chamberlain, Cray Inc

Chapel 1.15 will feature improvements to array memory management. These improvements significantly reduce the amount of leaked memory and also improve performance in some cases. This talk seeks to describe the progress in this area. First, this talk will describe the previous situation, the approach towards improvement, and the impact of progress in this area. Then, this talk will discuss in more detail language design issues that the effort has brought up.

Previous to this work, array memory management was incorrect and slow. These problems were a result of the original semantics of arrays which required reference counting. To make progress on this issue, we sought to improve the language and implementation to avoid reference counting where it was no longer necessary. In particular, the current lexical scoping rule removes the main reason to do array reference counting. The lexical scoping rule was introduced relatively recently and in particular it makes it an error to use a variable after the declaring scope exits. The result of this effort was a huge reduction in leaks and performance improvement in some cases.

This effort brought up several language design questions:
1) Do arrays return by value or by reference?
2) Do tuples capture arrays by reference or by value?
3) How does default array intent interact with sparse arrays of arrays?

The question of whether or not arrays return by value was important to settle because the previous strategy of returning arrays by reference relied upon reference counting in order to be correct. In particular, it should be possible to return local array variable, but such a variable might be deallocated when the function exits. Additionally, other types in the language return by value by default. We decided to make arrays return by default to enable progress on array memory management. That choice also has the side effect of making return behavior more consistent across types.

The choice of making arrays return by value had an impact on the semantics of tuples. Previously, tuples always captured arrays by reference. However, that would mean that returning a tuple of local arrays could refer to deallocated stack memory. Instead, we adjusted passing and returning tuples to match the semantics of the individual components. The result is that an array returned in a tuple is still returned by value; but an array in an argument tuple is still captured by reference.

Lastly, there is surprising interaction in the current implementation that was initially observed as a performance problem with the memory management for arrays. The language design elements that interact in a surprising way are a) that default intent for arrays is *ref* and b) the *ref-pair* feature (see 13.7.3 Choosing a Ref Return Intent Function Based on Calling Context of the Chapel language specification [1]). Let's consider an example in order to see the issue.

The first thing to know is that accessing an element of a sparse array is a fatal error if the access is a write to the element. For example,

```
{
  var dense = {1..10};
  var sps : sparse subdomain(dense);
  var A : [sps] int;

  var x = A[10]; // OK: this sets x to the sparse array's zero value
  A[10] = 10; // this is a runtime error : can't set the zero value
}
```

Now suppose that we implement a generic function to output an array element, like so:

```
proc writeElement(x) {
   writeln("element is ", x);
}
```

Let's apply this writeElement function to an array of arrays:

```
{
  var A:[1..10] [1..5] int;

  writeElement(A[1]);
}
```

This example works as expected and prints

```
element is 0 0 0 0 0
```

What happens if the outer array is a sparse array instead of a dense array?

```
{
  var dense = {1..10};
  var sps : sparse subdomain(dense);
  var A : [sps] [1..5] int;

  writeElement(A[1]);
}
```

Now the program halts with an error like this:

```
error: halt reached - attempting to assign a 'zero' value in a sparse
array: (1)
```

Since the default argument intent is *ref* for arrays, the compiler interprets the call writeElement(A[1]) as potentially setting the element A[1], which is an error since that element is the sparse array zero value. We will discuss how this issue might be solved.

We believe that solving these language design issues represents a significant improvement to the Chapel language and its implementation.

References

[1] Chapel Language Specification, version0.982, available at http://chapel.cray.com/docs/1.14/_downloads/chapelLanguageSpec.pdf