A Tutorial on Omega

The Omega calculator is a text-based interface to the Omega library, a set of C++ classes for manipulating integer tuple sets and relations. Its applications include dependence analysis, program transformations, and generating code from transformations.

1. Using the Omega Calculator

The Omega calculator is available on the myth cluster as /usr/class/cs243/omega/. The executable you need to use is /usr/class/cs243/omega/bin/oc. Create the following alias for convenience:

```
$ alias oc /usr/class/cs243/omega/bin/oc
```

You might want to include the above command in .cshrc or .bashrc file. To run the omega calculator on commands in the file filename.in, type

```
$ oc filename.in
```

If you prefer building your own Omega, you can get the sources by using git:

```
$ git clone git://github.com/davewathaverford/the-omega-project.git
```

And then, to build Omega:

```
$ cd omega
$ make depend
$ make all
```

Note that you can safely ignore the errors reported in the make depend process.

2. Using Omega Calculator for Affine Partitioning

The Omega library manipulates integer tuple sets and relations. This manipulation is useful for dependence analyses and affine transformations since we can view iteration spaces as integer tuple sets and affine transformations as integer tuple relations. For example, an integer tuple set

```math
\{[i,j] : 0 \leq i < n \&\& 0 \leq j < n\}
```

can represent the iteration space of a two-level loop, and an integer tuple relation

```math
\{[i, j] -> [j, i]\}
```

represents the two-dimensional transpose affine transformation.

Sets and relations are simplified before being displayed to the user. During simplification, it may be determined that a relation or a set contains no points or contains all points, in which case the simplified
result will be False or True, respectively. For example,
   \{[i] \to [i]\} \cap \{[i] \to [i + 1]\}

evaluates to
   \{[i] \to [i0] : False\}.

Although the Omega library can be used for many other purposes, this tutorial only discusses how it can be used to find affine partitionings with the maximum degree of communication-free parallelism. Note that the Omega library does not automate the entire procedure of the affine partitioning—some manual calculation is necessary.

The part below lists the steps needed to find an affine partitioning of a program.

1. Specify the affine partitioning constraints (boundary conditions and data dependences) for a given program.
2. Input the constraints into the Omega calculator to simplify them if necessary.
3. Manually solve the (simplified) affine constraints and get an affine partition that maximizes the degree of parallelism with no communication.
4. Input the affine partitioning to the Omega calculator and generate the code.

If you are interested in the details of the simplification procedure in the Omega library, refer to the documentation of the Omega calculator.

NOTE: All the example files are under directory /usr/class/cs243/omega/.

3. A simple example: re-indexing

Given the following re-indexing C Source Code (for more about re-indexing, refer to page 848 of the textbook):

```c
for (i = 1; i <= N; i++) {
    Y[i] = Z[i];  /* (s1) */
    X[i] = Y[i-1]; /* (s2) */
}
```

We want to find affine partition mappings \( p = C1*i1 + c1 \) and \( p = C2*i2 + c2 \), for statements \( s1 \) and \( s2 \), respectively, that parallelize the code without needing communication. The constraints are that statements referring to the same memory location must be mapped to the same processor. We first write affine equations that represent conditions for such mappings, simply the equations, and then solve the affine partition mapping problem.

(a) Write the affine equations

The followings are the affine equations for the condition that \( Y[i] \) in \( s1 \) refers to the same location as \( Y[i - 1] \) in \( s2 \) in the C code above:

1 <= i1 <= N;
1 <= i2 <= N;
i1 = i2 - 1;
(b) Simply the affine equations using the Omega calculator

We represent the above constraints, in a file called “domain.in”, as follows:

```plaintext
symbolic N;
D := {[i1,i2]: 1 <= i1 <= N && 1 <= i2 <= N && i1 = i2-1};
```

Line 1 says that N (the index bound) is a symbolic constant; line 2 defines an integer tuple set called D; line 3 says simplify D.

Running the omega calculator on domain.in

```plaintext
$ oc domain.in
```

produces the simplified constraints for i1 and i2:

```
{[i1,i1+1]: 1 <= i1 < N}
```

The results indicate that i2 = i1+1.

(c) Solve the affine equations to find an affine partition.

Let C1*i1+c1 and C2*i2+c2 be the affine partitionings for instruction s1 and s2, respectively. of interest.

No communication means adding the constraint

```
C1*i1 + c1 = C2*i2 + c2
```

to the result of (b). Refer to page 832 in the text book.

Substituting i2 with i1+1, we get

for all i1 such that 1 <= i1 <= N

```
C1*i1 + c1 = C2*(i1 + 1) + c2.
```

Since

```
C1*i1 = C2*i1 + (C2 + c2 - c1).
```

must hold true for all values of i1 satisfying the constraints. Therefore, C1 = C2 and C2+c2-c1 = 0.

The trivial answer of setting everything to 0 will result in a 0-rank partitioning, which is uninteresting.

The simplest non-trivial answer is: C1 = C2 = 1, c1 = 0, c2 = -1.

Thus, p = i1 for statement s1, and p = i2 - 1 for statement s2.

For an n-deep loop nest, there can be n possible independent solutions. The number of independent answers determines the rank of the affine partitionings, and hence the dimension of parallelism.

(c) Generate code from the affine partitions:

```plaintext
symbolic N;
S1 := {[p,i1]: p = i1 && 1 <= i1 <= N};
S2 := {[p,i2]: p = i2 - 1 && 1 <= i2 <= N};
codegen S1, S2;
```

Lines 2 and 3 are the equations that represent affine partition mappings and iteration boundaries. S1 is for statement 1 and S2 is for statement 2. Line 4 asks the Omega calculator to generate SPMD code.

Run the following command:
Then, the Omega calculate will show the following result:

```plaintext
$ oc cs243_1.in

# Omega Calculator v2.1 (based on Omega Library 2.1, July, 2008):
# symbolic N, M;
#
#
# S1 := {[p,i1]: p = i1 && 1 <= i1 <= N};
#
#
# S2 := {[p,i2]: p = i2 - 1 && 1 <= i2 <= N};
#
#
# codegen S1,S2;
if (N >= 1) {
  s2(0,1);
} for(t1 = 1; t1 <= N-1; t1++) {
  s1(t1,t1);
  s2(t1,t1+1);
} if (N >= 1) {
  s1(N,N);
}
```

“s1(t1, t1)” means substituting i with t1 in the first statement and “s2(t1,t+1)” means substituting i with t1+1 in the second statement to get the SPMD code (t1 is the processor number).

There is another example called cs243_2.in that you can try yourself. It does scaling (You can also refer to Page 848 of the textbook for this example).

### 4. A data dependence analysis example

The example above was deliberately chosen for its simplicity. Let’s see how the Omega calculator can help solve more complicated examples. Suppose that we have the following code:

```plaintext
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
    A[i + j][3*i + j] = A[2*i][5 - 2*j]; /* (s1) */
  }
}
```

The dependence constraint between the read and write operations is (in file domain_2.in):

```plaintext
symbolic N;
```
Applying the Omega calculator to the constraints returns that there is no dependence:

\[
\{ [i1, j1, i2, j2] : 0 <= i1, j1, i2, j2 < N \&\& i1 + j1 = 2*i2 \&\& 3*i1 + j1 = 5 - 2*j2 \}
\]

Therefor, with the assistance of the Omega calculator, we can safely conclude that there is no solution (thus no potential reference to the same memory location) to the affine equations.

5. Another affine partitioning example

This is the same as Figure 11.28 in page 839 of the text book. Note that this example is also structurally similar to the one illustrated in class (lecture note 13, slide 10). We have a nested loop and two statements.

```c
for (i = 1; i <= 100; i++) {
    for (j = 1; j <= 100; j++) {
        X[i, j] = X[i, j] + X[i - 1, j]; /* (s1) */
        Y[i, j] = Y[i, j] + X[i, j - 1]; /* (s2) */
    }
}
```

As shown in Example 11.44 of the text book (page 839), an affine partitioning mapping that maximizes the degree of parallelism with no communication is: p = i1 - j1 - 1 (for s1) and p = i2 - j2 (for s2).

Input this affine partition to the Omega calculator, and you will get the following result. Note that 1 <= i1 <= 100 && 1 <= j1 <= 100 can be succinctly written as 1 <= i1, j1 <= 100.

Also, notice that the Omega calculator does Figure 11.28 -> Figure 11.29 -> Figure 11.32 -> Figure 11.34(b) transformations for you! (Figure 11.34(b) is the same as the result shown below except for a few minor details including the first two and the last two lines in Figure 11.34(b).)

Result:

```c
# Omega Calculator v2.1 (based on Omega Library 2.1, July, 2008):
# S1 := {[p,i1,j1] : 1 <= i1,j1 <= 100 && p = i1-j1-1};
#
# S2 := {[p,i2,j2] : 1 <= i2,j2 <= 100 && p = i2-j2};
#
# codegen S1,S2;
for(t1 = -100; t1 <= 99; t1++) {
    if (t1 >= 0) {
        s2(t1,t1+1,1);
    }
    for(t2 = max(t1+2,1); t2 <= min(t1+100,100); t2++) {
        s1(t1,t2,t2-t1-1);
        s2(t1,t2,t2-t1);
    }
}
```
if (t1 <= -1) {
    s1(t1, t1+1, 100);
}
}