Barrier Inference

David Gay

May 14, 1997

SPMD Synchronization

SPMD programs use *barrier* as the main synchronization construct:
All processes must reach a barrier before any can proceed.

A typical SPMD program structure is:

```
work1();
barrier;
work2();
barrier;
work3();
```

Synchronization ensures: all processors do `work1` together, `work2` together, ...

Synchronization bugs

Synchronization bugs are hard to find.

A typical example of incorrect barrier synchronization is:

```
while (random()) barrier;
work1(); barrier;
work2(); barrier;
work3();
```

Two problems:

- Some processes terminate before others $\Rightarrow$ program hangs at a barrier
- Some processes execute `work1` while others are in `work2` or `work3` $\Rightarrow$ probably incorrect behaviour

Correct synchronization

Correct synchronization is not always trivial:

```
void f(int n) {
  for (int i = 0; i < n; i++) {
    work(i);
    barrier;
  }
}
```

This is correct if all processes call f with the same argument, because all processes go round the barrier-containing loop the same number of times.

Another example:

```
int x = broadcast(y);
if (x) barrier;
```

This is correct because all processes get the same value for x from the broadcast.

Barrier Inference

Definition: A program is correct if for every statement s, all processes that enter s exit s having executed the same sequence of synchronization operations.

Correct programs share the following characteristics:

- They contain a number of *replicated* variables, i.e. variables that exist independently on each processor but contain the same values
- The control structures with barriers depend on replicated variables.

Barrier Inference is an analysis that:

- Finds the replicated variables of a program
- Verifies the synchronization pattern

$\Rightarrow$ Barrier Inference guarantees the correctness of synchronization at **compile-time**.

Split-C Implementation

Program	Lines	Number of barriers	Function signatures	Single-valued globals	Single-valued casts	Other changes	Analysis time
bitonic	857	5	-	-	2	-	0.1s
cannon	501	17	1	-	-	-	0.3s
cg	453	18	3	-	-	-	0.1s
cholesky	1542	38	4	-	2	-	2.3s
column	651	7	1	-	-	-	0.1s
mm	508	23	1	-	-	1	0.2s
radix	379	7	-	-	2	-	0.1s
sample	302	9	-	-	-	-	0.1s
wator	348	10	-	3	-	2	0.1s
nbody	546	7	-	2	3	2	0.3s
em3d	1080	16	-	-	-	-	0.3s
barnes	2804	73	2	6	7	2	0.6s

Kernels:

- bitonic, column, sample, radix: Sorting programs.
- cannon: Matrix multiplication using Cannons algorithm.
- cg: Solves a set of equations using the conjugent gradient method.
- cholesky: Seven different implementations of Cholesky decomposition.
- mm: Matrix-multiply, blocked or unblocked.

Applications:

- wator: Simulation of particle-like fish under current.
- nbody: A simple n body simulation code.
- em3d: 3d electro-magnetic simulation.
- barnes: Simulate the interaction of a system of n bodies using the Barnes-Hut hierarchical method.

Titanium

Synchronization is important, its structure should be more explicit
$\Rightarrow$ Titanium requires the explicit declaration of replicated variables.

Advantages:

- Documents code
- The analysis is simpler, therefore faster

Allows replicated objects:

- The object exists on each processor, but has the same dynamic type everywhere
- The fields of the object are consistent everywhere

Titanium examples

The first example above would become:

```
void f(replicated int n) {
  for (replicated int i = 0; i < n; i++) {
    work(i);
    barrier;
  }
}
```

The other example is rewritten as:

```
replicated int x = broadcast(y);
if (x) barrier;
```

Summary

- Global synchronization bugs are tricky to find
- Barrier Inference eliminates all these bugs at **compile-time**.

  $\Rightarrow$ especially important for large systems, such as ASCI.
- Language extensions make the synchronization structure explicit, hence make the program clearer.