Notes on Loop Distribution and Parallelization

Mihail Georgiev

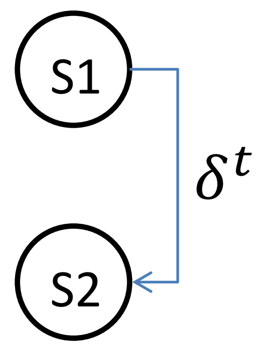
Loop Distribution

- separate one loop into two or more
- only okay if dependences are preserved

```
for (i = 1; i < n; i++) {
S1:  a[i] = b[i];
S2:  c[i] = a[i - 1];
}
```

-----vs.-----

```
for (i = 1; i < n; i++) {
S1:  a[i] = b[i];
}
for (i = 1; i < n; i++) {
S2:  c[i] = a[i - 1];
}
```



Loop Distribution

original loop

- loop-carried dependence
- not parallelizable

```
for (i = 1; i < n; i++) {
S1:   a[i] = b[i];
S2:   c[i] = a[i - 1];
}
```

distributed loop

- Loop-indep. dependence
- parallelizable

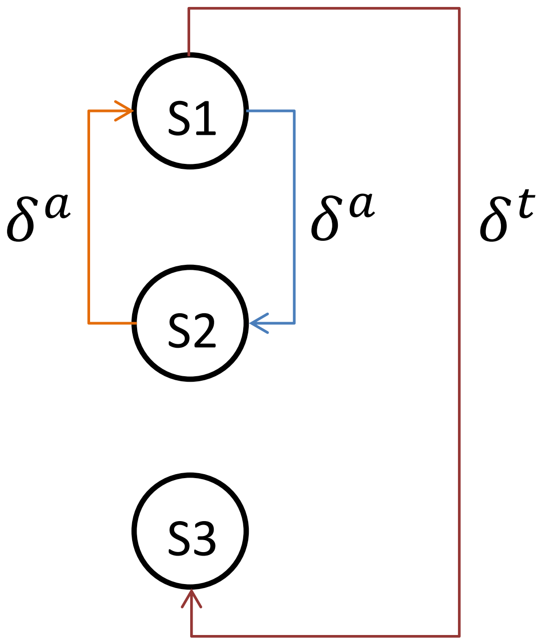
```
#pragma omp parallel for
for (i = 1; i < n; i++) {
S1:   a[i] = b[i];
}
#pragma omp parallel for
for (i = 1; i < n; i++) {
S2:   c[i] = a[i - 1];
}
```

Consider Progression

i	code
	`S1: B[0] = A[0];`
	`S2: A[0] = A[0] + B[1];`
	`S3: C[0] = 2 * B[0];`
	`S1: B[1] = A[1];`
	`S2: A[1] = A[1] + B[2];`
	`S3: C[1] = 2 * B[1];`
	`S1: B[2] = A[2];`
	`S2: A[2] = A[2] + B[3];`
	`S3: C[2] = 2 * B[2];`
	`S1: B[3] = A[3];`
	`S2: A[3] = A[3] + B[4];`
	`S3: C[3] = 2 * B[3];`

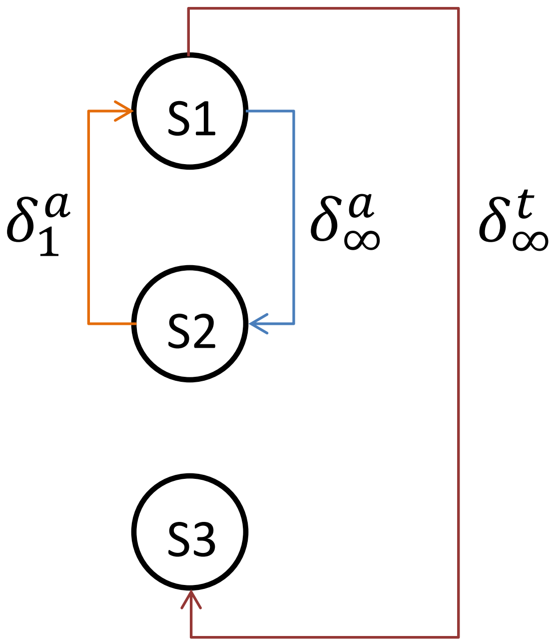
Consider Progression

i	code
	`S1: B[0] = A[0];`
	`S2: A[0] = A[0] + B[1];`
	`S3: C[0] = 2 * B[0];`
	`S1: B[1] = A[1];`
	`S2: A[1] = A[1] + B[2];`
	`S3: C[1] = 2 * B[1];`
	`S1: B[2] = A[2];`
	`S2: A[2] = A[2] + B[3];`
	`S3: C[2] = 2 * B[2];`
	`S1: B[3] = A[3];`
	`S2: A[3] = A[3] + B[4];`
	`S3: C[3] = 2 * B[3];`



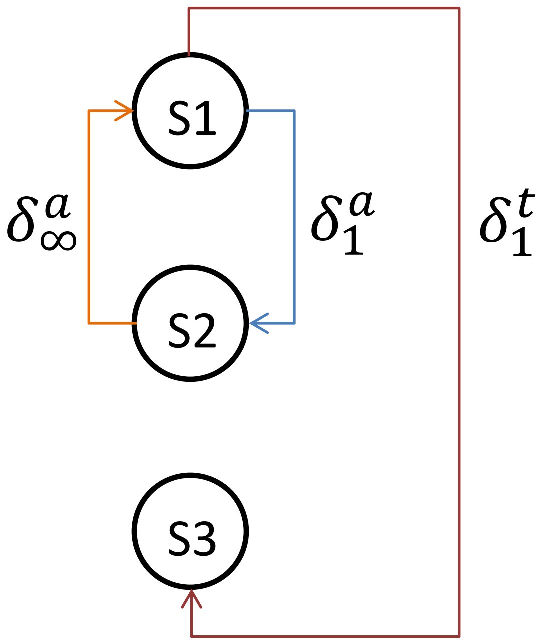
Loop Option 1

i	code
0	S1: B[0] = A[0];
	S2: A[0] = A[0] + B[1];
	S3: C[0] = 2 * B[0];
1	S1: B[1] = A[1];
	S2: A[1] = A[1] + B[2];
	S3: C[1] = 2 * B[1];
2	S1: B[2] = A[2];
	S2: A[2] = A[2] + B[3];
	S3: C[2] = 2 * B[2];
3	S1: B[3] = A[3];
	S2: A[3] = A[3] + B[4];
	S3: C[3] = 2 * B[3];



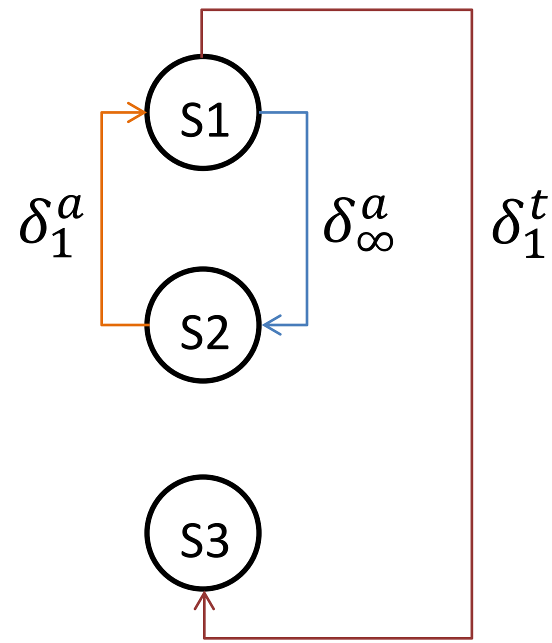
Loop Option 2

i	code
-1	`S1: B[0] = A[0];`
0	`S2: A[0] = A[0] + B[1];`
	`S3: C[0] = 2 * B[0];`
	`S1: B[1] = A[1];`
1	`S2: A[1] = A[1] + B[2];`
	`S3: C[1] = 2 * B[1];`
	`S1: B[2] = A[2];`
2	`S2: A[2] = A[2] + B[3];`
	`S3: C[2] = 2 * B[2];`
	`S1: B[3] = A[3];`
3	`S2: A[3] = A[3] + B[4];`
	`S3: C[3] = 2 * B[3];`



Loop Option 3

i	code
-1	`S1: B[0] = A[0];`
	`S2: A[0] = A[0] + B[1];`
0	`S3: C[0] = 2 * B[0];`
	`S1: B[1] = A[1];`
	`S2: A[1] = A[1] + B[2];`
1	`S3: C[1] = 2 * B[1];`
	`S1: B[2] = A[2];`
	`S2: A[2] = A[2] + B[3];`
3	`S3: C[2] = 2 * B[2];`
	`S1: B[3] = A[3];`
	`S2: A[3] = A[3] + B[4];`
4	`S3: C[3] = 2 * B[3];`

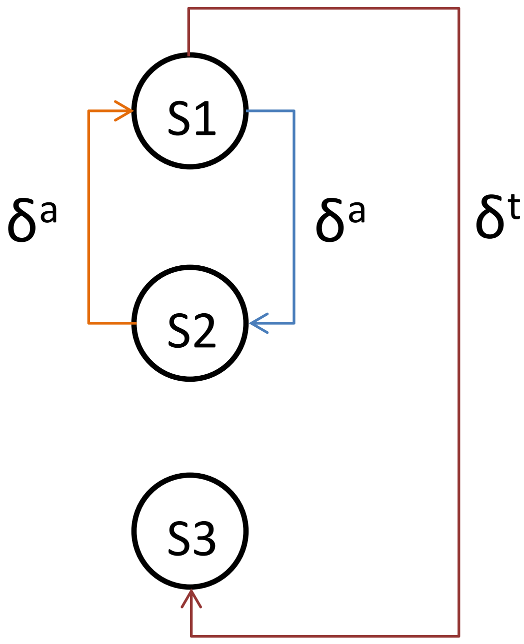


A Few Conclusions

- choose loop to have the most loop-independent (level ∞) dependencies
  - option 1 is best (it has 2 such dependencies)
- a cycle in the dependency graph means it is not possible to create only loop-independent dependencies
  - either $S1\ \delta^a\ S2$ or $S2\ \delta^a\ S1$ will be loop-carried

Graph Cycle ⇒ Unparallelizeable Code

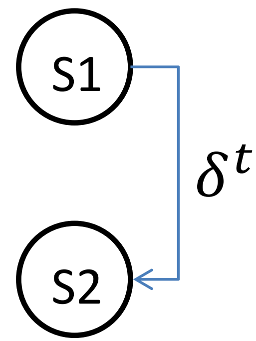
i	code
	`S1: B[0] = A[0];`
	`S2: A[0] = A[0] + B[1];`
	`S3: C[0] = 2 * B[0];`
	`S1: B[1] = A[1];`
	`S2: A[1] = A[1] + B[2];`
	`S3: C[1] = 2 * B[1];`
	`S1: B[2] = A[2];`
	`S2: A[2] = A[2] + B[3];`
	`S3: C[2] = 2 * B[2];`
	`S1: B[3] = A[3];`
	`S2: A[3] = A[3] + B[4];`
	`S3: C[3] = 2 * B[3];`



Loop Alignment

- removes loop-carried dependence

```
for (i = 1; i < n; i++) {
S1:  a[i] = b[i];
S2:  c[i] = a[i - 1];
}
```



-----vs.-----

```
for (i = 0; i < n; i++) {
S1:  if (i >= 1)     a[i] = b[i];
S2:  if (i < n - 1) c[i + 1] = a[i];
}
```

Loop Alignment: Extra Conditionals

- conditional statements slow down execution

```
     for (i = 0; i < n; i++) {
     S1:  if (i >= 1)     a[i] = b[i];
     S2:  if (i < n - 1) c[i + 1] = a[i];
     }
```

is slower than

```
S20: c[1] = a[0];
     for (i = 1; i < n - 1; i++) {
     S1:  a[i] = b[i];
     S2:  c[i + 1] = a[i];
     }
S1n: a[n] = b[n];
```

Runtime Comparison

```
                    ...
     start = omp_get_wtime();
     for (i = 0; i < n; i++) {
     S1:    if (i >= 1)   a[i] = b[i];
     S2:    if (i < n - 1)      c[i + 1] = a[i];
     }
     stop = omp_get_wtime();
     printf("extra ifs runtime: %g\n", stop - start);
                    ...
     start = omp_get_wtime();
S20: c[1] = a[0];
     for (i = 1; i < n - 1; i++) {
     S1i:   a[i] = b[i];
     S2i:   c[i + 1] = a[i];
     }
S1n: a[n] = b[n];
     stop = omp_get_wtime();
     printf("peeled runtime:    %g\n", stop - start);
                    ...
```

Runtime Comparison

- for n = 100 000 000, output is

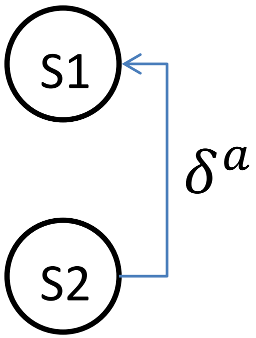
```
extra ifs runtime: 1.11077
peeled runtime:    0.727978
```

- peeling off the conditional statements is easy and significantly beneficial

Loop Alignment: Statement Reordering

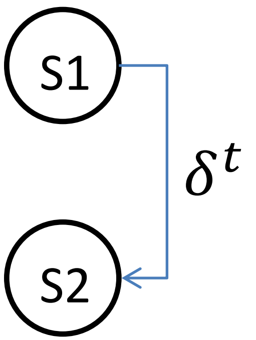
- alignment may not immediately work

```
for (i = 0; i < n - 1; i++) {
S1:  a[i] = b[i];
S2:  c[i] = a[i + 1];
}
```



-----vs.-----

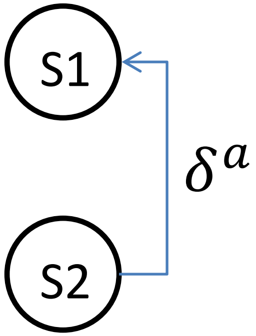
```
for (i = 0; i < n; i++) {
S1:  if (i < n - 1) a[i] = b[i];
S2:  if (i >= 1)    c[i - 1] = a[i];
}
```



Loop Alignment: Statement Reordering

- interchanging **`S1`** and **`S2`** fixes that

```
for (i = 0; i < n - 1; i++) {
S1:  a[i] = b[i];
S2:  c[i] = a[i + 1];
}
```



-----vs.-----

```
for (i = 0; i < n; i++) {
S2:  if (i >= 1)     c[i - 1] = a[i];
S1:  if (i < n - 1) a[i] = b[i];
}
```