Eliminating loop dependencies

Troels Henriksen
Based on material by Cosmin Oancea

A seemingly sequential loop

```
float tmp;
for (int i = 2; i < N; i++) {
  S1: tmp = 2 * B[i-2];
  S2: A[i] = tmp;
  S3: B[i] = tmp + B[i-1];
}
```

Dependencies:

- $S_3 \rightarrow S_1$ with direction < due to `B`.
- $S_1 \rightarrow S_3$ with direction = due to `tmp`.
- $S_1 \rightarrow S_2$ with direction = due to `tmp`.

A seemingly sequential loop

```
float tmp;
for (int i = 2; i < N; i++) {
  S1: tmp = 2 * B[i-2];
  S2: A[i] = tmp;
  S3: B[i] = tmp + B[i-1];
}
```

Dependencies:

- $S_3 \rightarrow S_1$ with direction < due to `B`.
- $S_1 \rightarrow S_3$ with direction = due to `tmp`.
- $S_1 \rightarrow S_2$ with direction = due to `tmp`.

Loop distribution

A transformation that factors out some statements of a loop, typically to improve locality or parallelism.

```
float tmp;
for (int i = 2; i < N; i++) {
  S1 : tmp = 2 * B[i-2];
  S2 : A[i] = tmp;
  S3 : B[i] = tmp + B[i-1];
}
```

- We distribute S_2 because it is not in a dependency cycle.

```
float tmp;
for (int i = 2; i < N; i++) {
  S1: tmp = 2 * B[i-2];
  S3: B[i] = tmp + B[i-1];
}
for (int i = 2; i < N; i++) {
  S2: A[i] = tmp;
}
```

- We distribute S_2 because it is not in a dependency cycle.
- But we have a problem, because we will not be reading the right `tmp` value. We must use *array expansion* to save all the values of `tmp` we will need.

```
float tmp[N];
for (int i = 2; i < N; i++) {
  S1 : tmp[i] = 2 * B[i-2];
  S3 : B[i] = tmp[i] + B[i-1];
}
for (int i = 2; i < N; i++) {
  S2 : A[i] = tmp[i];
}
```

- We distribute S_2 because it is not in a dependency cycle.
- But we have a problem, because we will not be reading the right `tmp` value. We must use *array expansion* to save all the values of `tmp` we will need.
- Instead of a single sequential loop, we now have a sequential loop and a parallel loop, and a bunch of extra memory usage. Worth it? Depends.

Eliminating false dependencies

False dependencies

Anti and output dependencies are often referred to as *false dependencies* because they can be eliminated in most cases by copying or privatisation.

Eliminating false dependencies

False dependencies

Anti and output dependencies are often referred to as *false dependencies* because they can be eliminated in most cases by copying or privatisation.

Anti dependency (WAR)

```
... = X
X   = ...
```

- Often a read from original array element followed by later update.
- Eliminate by copying original array.

Eliminating false dependencies

False dependencies

Anti and output dependencies are often referred to as *false dependencies* because they can be eliminated in most cases by copying or privatisation.

Anti dependency (WAR)

```
...  = X
X    = ...
```

- Often a read from original array element followed by later update.
- Eliminate by copying original array.

Output dependency (WAW)

```
X = ...
X = ...
```

- Often the case that every read is covered by a write in same iteration.
- Can often be fixed by privatisation.

Eliminating WAR dependencies by copying

```
float tmp = A[0];
for (int i=0; i<N-1; i++)
    A[i] = A[i+1];
A[N-1] = tmp;
```

Eliminating WAR dependencies by copying

```c
float tmp = A[0];
for (int i=0; i<N-1; i++)
    A[i] = A[i+1];
A[N-1] = tmp;
```

$\Rightarrow$

```c
float Acopy[N];
#pragma omp parallel for
for (int i=0; i<N; i++)
    Acopy[i] = A[i];
}
tmp = A[0];
#pragma omp parallel for
for (int i=0; i<N-1; i++)
    A[i] = Acopy[i+1];
A[N-1] = tmp;
```

Eliminating WAW dependencies by privatisation

```
int A[M];
for (int i=0; i<N; i++) {
  for (int j=0, j<M; j++) // Overwrites all of A
    A[j] = (4*i+4*j) % M;
  for (int k=0; k<N; k++)
    X[i][k] = X[i][k-1] * A[ A[(2*i+k)%M] % M];
}
```

- Dependencies on `A` in all directions (∗).
- Reads from `A` un-analysable because of the indirect access `A[A[...]]`.
- **But there is hope for parallelising this!**

Eliminating WAW dependencies by privatisation

```
int A[M];
for (int i=0; i<N; i++) {
  for (int j=0, j<M; j++) // Overwrites all of A
    A[j] = (4*i+4*j) % M;
  for (int k=0; k<N; k++)
    X[i][k] = X[i][k-1] * A[ A[(2*i+k)%M] % M];
}
```

- Dependencies on `A` in all directions (∗).
- Reads from `A` un-analysable because of the indirect access `A[A[...]]`.
- **But there is hope for parallelising this!**
  - All reads from `A` must necessarily access values written in the same (outer) iteration.
  - So we can give each (outer) iteration its own copy of A!

Privatising A

```
int A[M];
for (int i=0; i<N; i++) {
  for (int j=0, j<M; j++)
    A[j] = (4*i+4*j) % M;
  for (int k=0; k<N; k++)
    X[i][k] = X[i][k-1] * A[ A[(2*i+k)%M] % M];
}
```

- Make room for `N` copies of `A` and give each iteration its own.

Privatising A

```
int A_expanded[N*M];
#pragma omp parallel for
for (int i=0; i<N; i++) {
  int *A = &A_expanded[i*M];
  for (int j=0, j<M; j++)
    A[j] = (4*i+4*j) % M;
  for (int k=0; k<N; k++)
    X[i][k] = X[i][k-1] * A[ A[(2*i+k)%M] % M];
}
```

- Make room for `N` copies of `A` and give each iteration its own.
- This may use a *lot* of memory—we really need only a copy *per thread*...

Privatising A

```
int A[M];
#pragma omp parallel for lastprivate(A)
for (int i=0; i<N; i++) {
  for (int j=0, j<M; j++)
    A[j] = (4*i+4*j) % M;
  for (int k=0; k<N; k++)
    X[i][k] = X[i][k-1] * A[ A[(2*i+k)%M] % M];
}
```

- Make room for `N` copies of `A` and give each iteration its own.
- This may use a *lot* of memory—we really need only a copy *per thread*...
- The OpenMP `lastprivate` clause automatically creates the per-thread private copies we need.

Summary

- Don't give up hope if a loop seems sequential.
- *Loop distribution* can let us extract parallel parts from partially sequential loops.
- *Array expansion* is sometimes needed to make this work.
- *Copying* can sometimes be used to eliminate WAR dependencies.
- *Privatisation* can sometimes be used to eliminate WAW dependencies.
- Making code more parallel often adds *overhead* and usually requires *more memory*—**always measure the effect of your changes**.