Parallelizing Compilers

Presented by Yiwei Zhang

Reference Paper

- Rudolf Eigenmann and Jay Hoeflinger, " **Parallelizing and Vectorizing Compilers**" , *Purdue Univ. School of ECE, High-Performance Computing Lab. ECE-HPCLab-99201*, January 2000

Importance of parallelizing

- With the rapid development of multi-core processors, parallelized programs can take such advantage to run much faster than serial programs
- Compilers created to convert serial programs to run in parallel are parallelizing compilers

Role of parallelizing compilers

- Attempt to relieve the programmer from dealing with the machine details
- They allow the programmer to concentrate on solving the object problem, while the compiler concerns itself with the complexities of the machine

Role of parallelizing compilers

- Much more sophisticated analysis is required of the compiler to generate efficient machine code for different types of machines

How to Parallelizing a Program

- Find the dependency in the program
- Try to avoid or eliminate the dependency
- Reduce overhead cost

Dependence Elimination and Avoidance

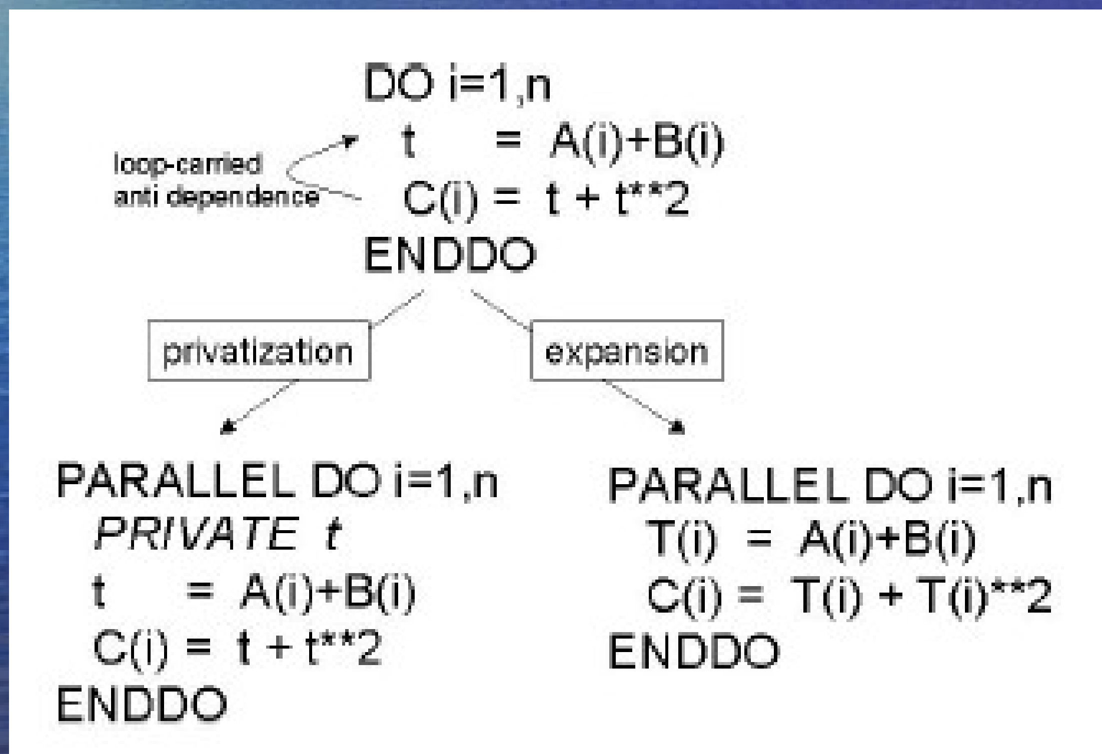
- A data dependence between two sections of a program indicates that during execution those two sections of code must be run in certain order
- anti dependence: READ before WRITE
- flow dependence: WRITE before READ
- output dependence: WRITE before WRITE

Data Privatization and Expansion

- Data privatization can remove anti and output dependences.
- These dependences are not due to computatio n having to wait for data values produced by o thers. Instead, it waits because it wants to assi gn a value to a variable that is still in use
- The basic idea is to use a new storage location so that the new assignment does not overwrite the old value too soon

Data Privatization and Expansion

- Example:



Idiom Recognition: Inductions, Reductions, Recurrences

- These transformations can remove true dependence, e.g., flow dependence
- This is the case where one computation has to wait for another to produce a needed data value
- The elimination is only possible if we can express the computation in a different way

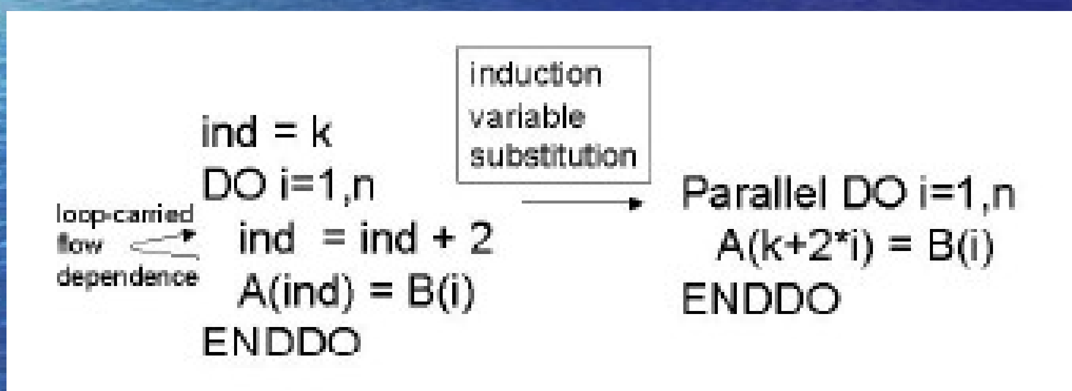
Induction Variable Substitution

- Finds variables which form arithmetic and geometric progressions which can be expressed as functions of the indices of enclosing loops
- Replaces these variables with the expressions involving loop indices

Induction Variable Substitution

- Example:

We can replace variable 'ind' with an arithmetic form of indice i



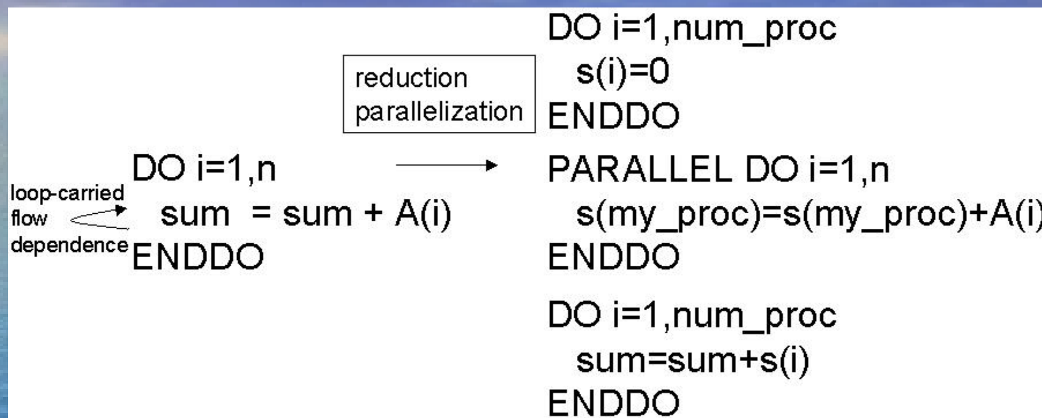
Identification of induction variables

- Through pattern matching (e.g., the compiler finds statements that modify variables in the described way)
- Through abstract interpretation (identifying the sequence of values assumed by a variable in a loop)

Reductions

- Reduction operations abstract the values of an array into a form with lesser dimensionality.
- The typical example is an array being summed up into a scalar variable

Reductions

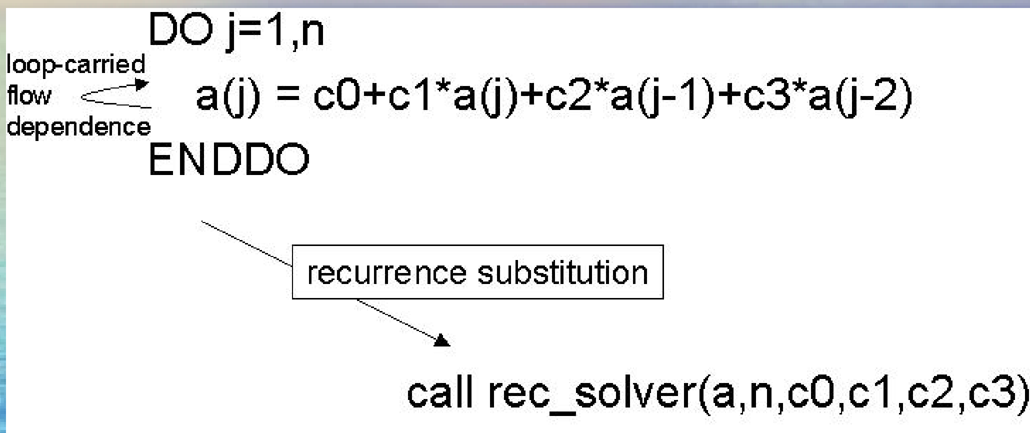


- We can split the array into p parts, sum them up individually by different processors, and then combine the results.
- The transformed code has two additional loops, for initializing and combining the partial results.

Recurrence

- Recurrences use the result of one or several previous loop iterations for computing the value of next iteration
- However, for certain forms of linear recurrences, algorithms are known that can be parallelized

Recurrence



- Compiler has recognized a pattern of linear recurrences for which a parallel solver is known. The compiler then replaces this code by a call to a mathematical library that contains the corresponding parallel solver algorithm

Parallel Loop Restructuring

- A parallel computation usually incurs an overhead when starting and terminating
- The larger the computation in the loop, the better this overhead can be amortized
- Use techniques such as loop fusion, loop coalescing, and loop interchange to optimize performance

Loop fusion

- Loop fusion combines two adjacent loops into a single loop.

```
PARALLEL DO i=1,n
  A(i) = B(i)
ENDDO

PARALLEL DO i=1,n
  C(i) = A(i)+D(i)
ENDDO
```

loop fusion →

```
PARALLEL DO i=1,n
  A(i) = B(i)
  C(i) = A(i)+D(i)
ENDDO
```

Loop coalescing

- Loop coalescing merges two nested loops into a single loop

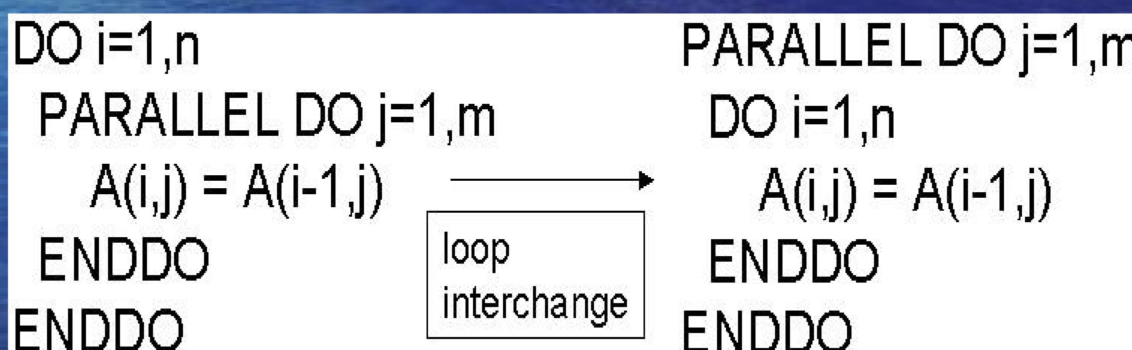
```
PARALLEL DO i=1,n
  DO j=1,m
    A(i,j) = B(i,j)
  ENDDO
ENDDO
```

loop coalescing →

```
PARALLEL DO ij=1,n*m
   i = 1 + (ij-1) DIV m
   j = 1 + (ij-1) MOD m
  A(i,j) = B(i,j)
ENDDO
```

Loop interchange

- Move an inner parallel loop to an outer position in a loop nest
- As a result, overhead is only incurred on ce



Questions?