Compile-time Parallelization of Subscripted Subscript Patterns

-Akshay Bhosale (akshay@udel.edu)
- Rudolf Eigenmann (eigenman@udel.edu)

Video presentation at : subscripted-subscript.akshayud.me
(scroll to the bottom of the page)

University of Delaware

The pattern and the problem

- Patterns wherein an array appears at the subscript of another array (e.g. a[b[i]])
- Write-accesses to an array with a subscript array across iterations of a *for*-loop create dependencies
- Molecular Dynamics Simulations and Adaptive Mesh Refinement are some of the scientific applications comprising of loops with such patterns
- Current compiler technology lacks the capabilities to analyze and automatically parallelize such loops

Approach to proving parallelism

- Defining subscript array properties that help prove non-overlap
- Analyzing loops that modify the content of the subscript array in program order
- Using an algorithm based on aggregation to determine the effect of the loop on the subscript array
- Proving the existence of a property based on aggregation

Important Properties: Monotonicity

- **Non-Strict Monotonicity**: Array ‘a’ is monotonically increasing if a[i] ≤ a[j] , for all i < j.
- E.g. From CG benchmark (NPB 3.3):

```
for (j = 0 ; j < n ; j++) {
    for (k = row[j] ; k < row[j+1] ; k++) {
        colidx[k] = colidx[k] - firstcol;
    }
}
```

- Outer loop can be parallelized if ‘row’ is monotonically increasing.

- **Strict Monotonicity**: Array ‘a’ is strictly monotonically increasing if a[i] < a[j] , for all i < j.
- From UA benchmark (NPB 3.3)

```
for (index = 0 ; index < n ; index++){
  k = action[index];
  nelt = nelttemp+(front[k]-1)*7;
  for( i = 0 ; i < 7 ; i++){
      tree[nelt + i] = ntemp + (( i + 1) % 8);
  }
}
```

- Outer loop is parallelizable if the expression for ‘nelt’ is strictly monotonically increasing.

Important Properties: Injectivity

- **Injective subscript array** : Every element of the subscript array is unique.
  - E.g. From UA benchmark (NPB 3.3)

```
for (miel = 0 ; miel < nelt ; miel++) {
        iel = mt_to_id[miel];
        id_to_mt[iel] = miel;
}
```

- Loop can be parallelized if array ‘mt_to_id’ is proven to be injective before the loop.

- **Injective Expression** : The subscript array is part of an expression and the entire expression is injective.
- E.g. From UA benchmark (NPB 3.3)

```
for (index = 0 ; index < n ; index++){
  k = action[index];
  nelt = nelttemp+(front[k]-1)*7;
  for( i = 0 ; i < 7 ; i++){
      tree[nelt + i] = ntemp + (( i + 1) % 8);
  }
}
```

- Expression for ‘nelt’ is strictly monotonic i.e. injective.

Analysis Algorithm based on Aggregation

- Determine subscript array properties by analyzing loops in program order
- Steps:
  1. In a loop nest, start with the innermost loop
  2. **Phase 1**: Symbolically analyze the loop body and determine the effect of one iteration on the variable's value i.e. determine the value range

     Phase 2: Aggregate for the entire iteration space
  3. Replace the loop with the aggregated expressions
  4. Analyze the next outer loop
  5. Repeat until all loops in the nest have been analyzed

Aggregation Example based on CG (NPB3.3)

```
for (i = 0 ; i < ROWLEN+1 ; i++){
        if(i == 0 ){
           j1 = i;
        }
        else{
           j1 = rowptr[i-1];
        }
        for(j = j1 ; j < rowptr[i] ; j++){
            product_array[j] = value[j] * vector[j];
        }
 }
```

Value of 'rowptr' comes at the subscript of 'product_array'. Outer loop can be parallelized if 'rowptr' is monotonic.

Proving Monotonicity of ‘rowptr’

```
for (i = 0 ; i < ROWLEN ; i++){
        count = 0;
        for (j = 0 ; j < COLUMNLEN ; j++){
            if(a[i][j] != 0){
                count++;
                column_number[index++] = j;
                value[ind++] = a[i][j];
            }
        }
        rowsize[i] = count;
    }
```

Loop A

```
rowptr[0] = 0;
for (i = 1 ; i < ROWLEN + 1 ; i++){
    rowptr[i] = rowptr[i-1] + rowsize[i-1];
}
```

Loop B

Aggregation: Loop A

```
for (i = 0 ; i < ROWLEN ; i++){
    count = 0;
  for (j = 0 ; j < COLUMNLEN ; j++){
    if( a[i][j] != 0){
      count++;
      column_number[index++] = j;
      value[ind++] = a[i][j];
    }
  }
  rowsize[i] = count;
}
```

- Phase 1 (inner loop):
  - **count : [λ : λ +1]**
  - column_number : ⊥
  - value : ⊥
- Phase 2 (inner loop):
  - **count : [Λ : Λ + COLUMNLEN]**
  - column_number : ⊥
  - value : ⊥

where,

λ : Value of the variable at the beginning of the iteration

Λ : Value of the variable at the beginning of the loop

⊥ : Unknown value

Aggregation: Loop A

```
for (i = 0 ; i < ROWLEN ; i++){
      count = 0;
   {
        count : [Λ : Λ + COLUMNLEN]
        column_number : ⊥
        value : ⊥
   }
    rowsize[i] = count;
}
```

- Inner loop replaced by the aggregated variable expressions

- Phase 1 (outer loop):
  - count : [0 : COLUMNLEN]
  - rowsize : [i] , [0 : COLUMNLEN]
- Phase 2 (outer loop):
  - count : [0 : COLUMNLEN]
  - rowsize : [0 : ROWLEN-1] , [0:COLUMNLEN]

index range value range

- At this point, 'rowsize' is a positive array.

Aggregation: Loop B

```
rowptr[0] = 0;
for (i = 1 ; i < ROWLEN + 1 ; i++){
    rowptr[i] = rowptr[i-1] + rowsize[i-1];
}
```

- Value of ‘rowptr’ in the previous iteration is added to the corresponding value of ‘rowsize’ and assigned to ‘rowptr’ in the current iteration.

- Phase 1 :

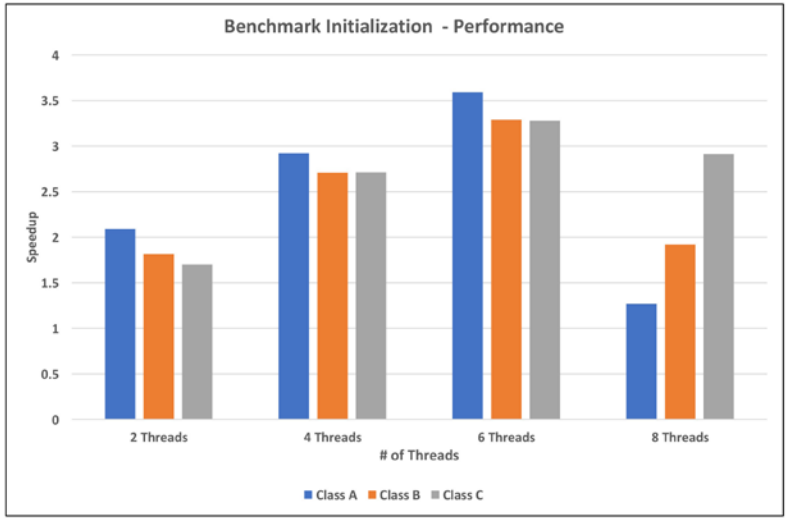
  rowptr : [i] , rowptr[i-1] + [0 : COLUMNLEN]

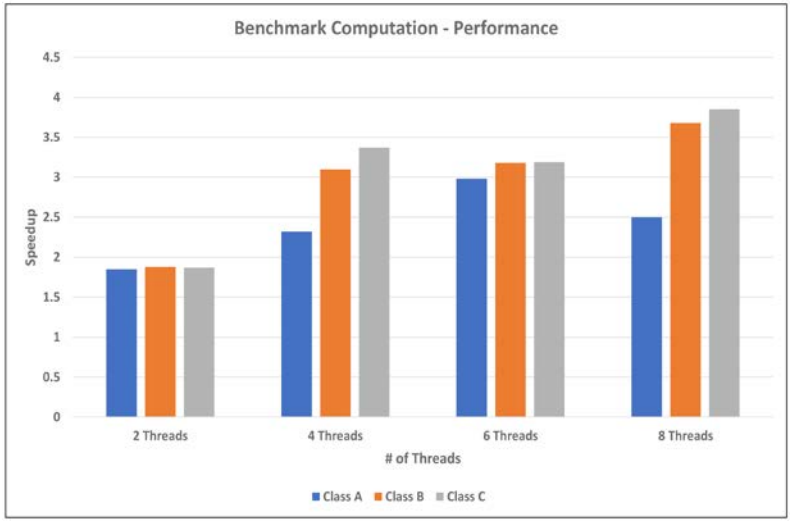
- Phase 2 :

  rowptr : [0 : ROWLEN] , **Monotonic_inc**

- Based on the recurrence relationship and aggregation, ‘rowptr’ has been proved to be Monotonically increasing.

Performance Results – CG (NPB 3.3)





Conclusions and Future work

- Novel compile-time analysis technique presented, powerful enough to parallelize a class of programs
- Applying the technique by hand gives tremendous speedup for the CG benchmark
- Next step is to apply the algorithm to even more scientific applications
- Developing the full aggregation algebra and implementing the extended Range Test in the Cetus source to source compiler