Workshop on Essential Abstractions in GCC

Parallelization and Vectorization in GCC

GCC Resource Center
(www.cse.iitb.ac.in/grc)

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

3 July 2012

Outline

- Transformation for parallel and vector execution
- Data dependence
- Auto-parallelization and auto-vectorization in Lambda Framework
- Conclusion

The Scope of This Tutorial

- What this tutorial does not address
  - Details of algorithms, code and data structures used for parallelization and vectorization
  - Machine level issues related to parallelization and vectorization
- What this tutorial addresses
  - GCC's approach of discovering and exploiting parallelism
  - Illustrated using carefully chosen examples

Part 1

Transformations for Parallel and Vector Execution

Vectorization: SISD ⇒ SIMD

- Parallelism in executing operation on shorter operands (8-bit, 16-bit, 32-bit operands)
- Existing 32 or 64-bit arithmetic units used to perform multiple operations in parallel
  A 64 bit word ≡ a vector of 2×(32 bits), 4×(16 bits), or 8×(8 bits)

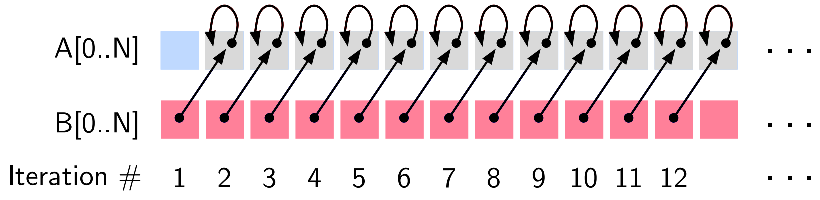
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

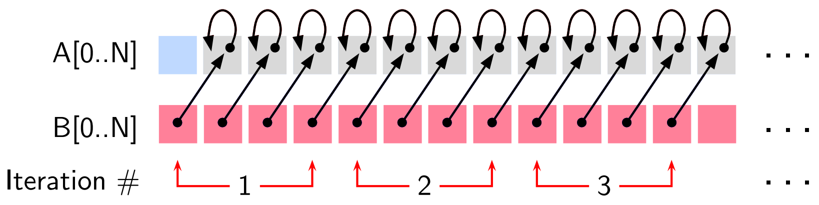
Vectorization Factor

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Vectorized Code

```
int A[N], B[N], i;
for (i=1; i<N; i=i+4)
  A[i:i+3] = A[i:i+3] + B[i-1:i+2];
```



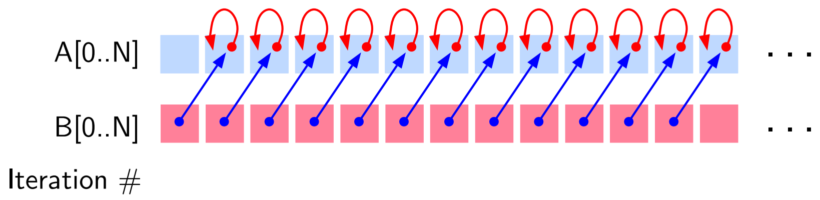
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



Example 1

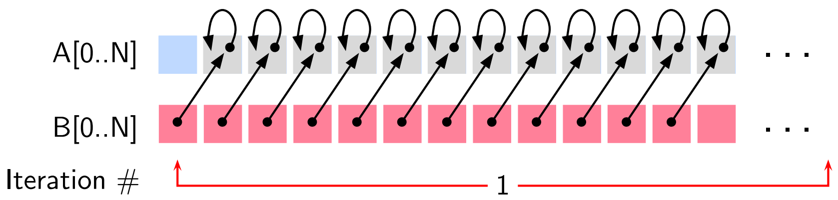
Vectorization	(SISD ⇒ SIMD)	: Yes
Parallelization	(SISD ⇒ MIMD)	: Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Parallelized Code

```
int A[N], B[N], i;
for-all (i=1 to N)
  A[i] = A[i] + B[i-1];
```

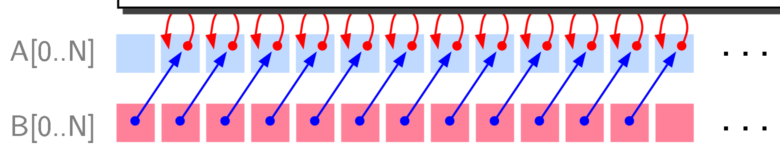


Example 1: The Moral of the Story

Vectorization	(SISD ⇒ SIMD)	: Yes
Parallelization	(SISD ⇒ MIMD)	: Yes

When the same location is accessed across different iterations, the order of reads and writes must be preserved

Nature of accesses in our example		
Iteration i	Iteration $i+k$	Observation
Read	Write	No
Write	Read	No
Write	Write	No
Read	Read	Does not matter



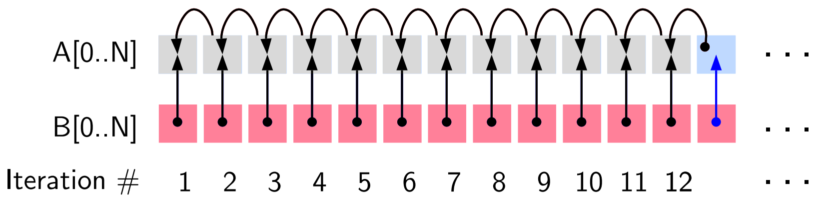
Example 2

Vectorization	(SISD ⇒ SIMD)	: Yes
Parallelization	(SISD ⇒ MIMD)	: No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



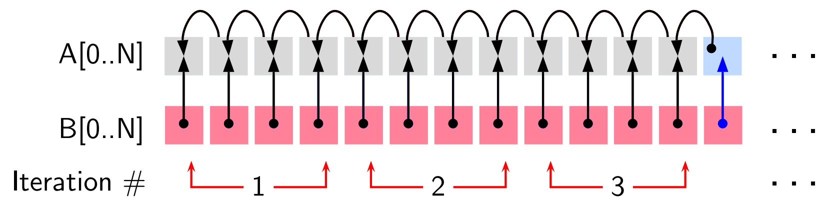
Example 2

Vectorization	(SISD ⇒ SIMD)	: Yes
Parallelization	(SISD ⇒ MIMD)	: No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

- Vector instruction is synchronized: All reads before writes in a given instruction
- Read-writes across multiple instructions executing in parallel may not be synchronized



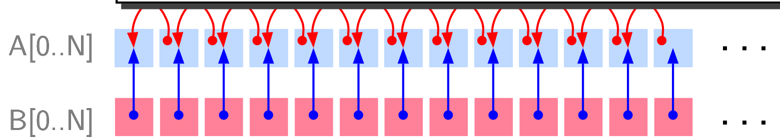
Example 2: The Moral of the Story

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : **No**

```
int A[
for (i
  A[i]
```

When the same location is accessed across different iterations, the order of reads and writes must be preserved

Nature of accesses in our example		
Iteration i	Iteration $i+k$	Observation
Read	Write	Yes
Write	Read	No
Write	Write	No
Read	Read	Does not matter

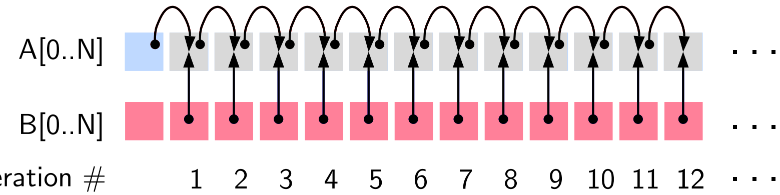


Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

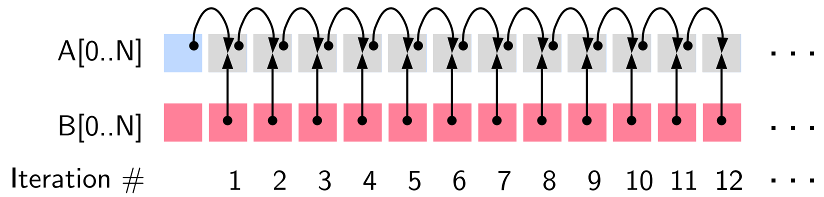


Example 3

Vectorization (SISD ⇒ SIMD) : **No**
Parallelization (SISD ⇒ MIMD) : **No**

```
int A[N], I
for (i=0; i
  A[i+1] =
```

Nature of accesses in our example		
Iteration i	Iteration $i+k$	Observation
Read	Write	No
Write	Read	Yes
Write	Write	No
Read	Read	Does not matter



Example 4

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : Yes

- This case is not possible
- Vectorization is a limited granularity parallelization
- If parallelization is possible then vectorization is trivially possible

Data Dependence

Let statements S_i and S_j access memory location m at time instants t and $t + k$

Access in S_i	Access in S_j	Dependence	Notation
Read m	Write m	Anti (or Pseudo)	$S_i \ \bar{\delta} \ S_j$
Write m	Read m	Flow (or True)	$S_i \ \delta \ S_j$
Write m	Write m	Output (or Pseudo)	$S_i \ \delta^o \ S_j$
Read m	Read m	Does not matter	

- Pseudo dependences may be eliminated by some transformations
- True dependence cannot be eliminated

Data Dependence

Consider dependence between statements S_i and S_j in a loop

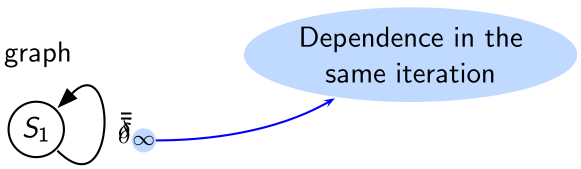
- Loop independent dependence. t and $t + k$ occur in the same iteration of a loop
  - S_i and S_j must be executed sequentially
  - Different iterations of the loop can be parallelized
- Loop carried dependence. t and $t + k$ occur in the different iterations of a loop
  - Within an iteration, S_i and S_j can be executed in parallel
  - Different iterations of the loop must be executed sequentially
- S_i and S_j may have both loop carried and loop independent dependences

Dependence in Example 1

- Program

```
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];   /* S1 */
```

- Dependence graph



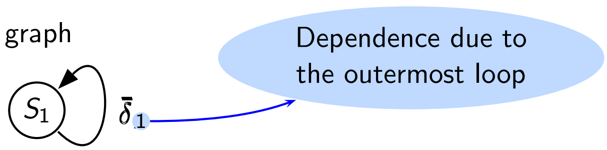
- No loop carried dependence
  Both vectorization and parallelization are possible

Dependence in Example 2

- Program

```
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i] = A[i+1] + B[i];   /* S1 */
```

- Dependence graph



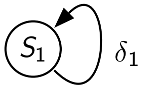
- Loop carried anti-dependence
  Parallelization is not possible
  Vectorization is possible since all reads are done before all writes

Dependence in Example 3

- Program

```
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];   /* S1 */
```

- Dependence graph



- Loop carried flow-dependence
  Neither parallelization not vectorization is possible

Iteration Vectors and Index Vectors: Example 1

```
for (i=0, i<4; i++)
  for (j=0; j<4; j++)
  {
     a[i+1][j] = a[i][j] + 2;
  }
```

Loop carried dependence exists if

- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

Conclusion: Dependence exists

Iteration Vector	Index Vector	
	LHS	RHS
0, 0	1, 0	0, 0
0, 1	1, 1	0, 1
0, 2	1, 2	0, 2
0, 3	1, 3	0, 3
1, 0	2, 0	1, 0
1, 1	2, 1	1, 1
1, 2	2, 2	1, 2
1, 3	2, 3	1, 3
2, 0	3, 0	2, 0
2, 1	3, 1	2, 1
2, 2	3, 2	2, 2
2, 3	3, 3	2, 3
3, 0	4, 0	3, 0
3, 1	4, 1	3, 1
3, 2	4, 2	3, 2
3, 3	4, 3	3, 3

Iteration Vectors and Index Vectors: Example 2

```
for (i=0, i<4; i++)
  for (j=0; j<4; j++)
  {
     a[i][j] = a[i][j] + 2;
  }
```

Loop carried dependence exists if

- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

Conclusion: No dependence

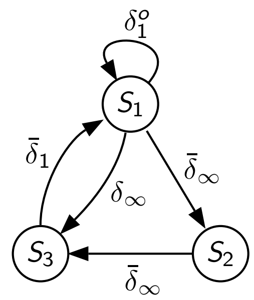
Iteration Vector	Index Vector	
	LHS	RHS
0, 0	0, 0	0, 0
0, 1	0, 1	0, 1
0, 2	0, 2	0, 2
0, 3	0, 3	0, 3
1, 0	1, 0	1, 0
1, 1	1, 1	1, 1
1, 2	1, 2	1, 2
1, 3	1, 3	1, 3
2, 0	2, 0	2, 0
2, 1	2, 1	2, 1
2, 2	2, 2	2, 2
2, 3	2, 3	2, 3
3, 0	3, 0	3, 0
3, 1	3, 1	3, 1
3, 2	3, 2	3, 2
3, 3	3, 3	3, 3

Example 4: Dependence

Program to swap arrays

```
for (i=0; i<N; i++)
{
    T = A[i];         /* S1 */
    A[i] = B[i];      /* S2 */
    B[i] = T;         /* S3 */
}
```

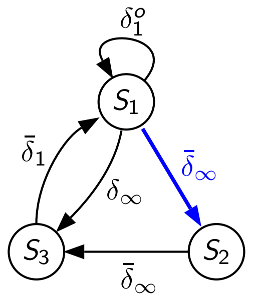
Dependence Graph



Example 4: Dependence

Program to swap arrays	Dependence Graph

```
for (i=0; i<N; i++)
{
    T = A[i];        /* S1 */
    A[i] = B[i];     /* S2 */
    B[i] = T;        /* S3 */
}
```

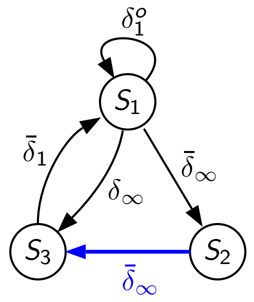


Loop independent anti dependence due to `A[i]`

Example 4: Dependence

Program to swap arrays	Dependence Graph

```
for (i=0; i<N; i++)
{
    T = A[i];        /* S1 */
    A[i] = B[i];     /* S2 */
    B[i] = T;        /* S3 */
}
```

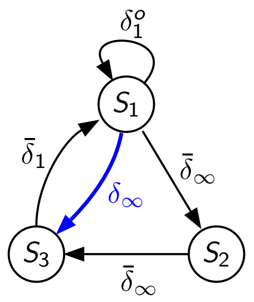


Loop independent anti dependence due to `B[i]`

Example 4: Dependence

Program to swap arrays	Dependence Graph

```
for (i=0; i<N; i++)
{
    T = A[i];        /* S1 */
    A[i] = B[i];     /* S2 */
    B[i] = T;        /* S3 */
}
```

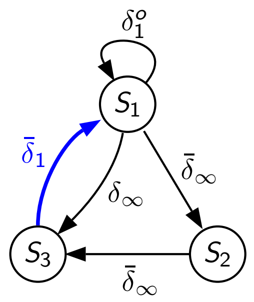


Loop independent flow dependence due to `T`

Example 4: Dependence

Program to swap arrays	Dependence Graph

```
for (i=0; i<N; i++)
{
    T = A[i];        /* S1 */
    A[i] = B[i];     /* S2 */
    B[i] = T;        /* S3 */
}
```

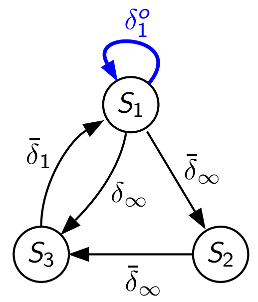


Loop carried anti dependence due to `T`

Example 4: Dependence

Program to swap arrays	Dependence Graph

```
for (i=0; i<N; i++)
{
    T = A[i];        /* S1 */
    A[i] = B[i];     /* S2 */
    B[i] = T;        /* S3 */
}
```

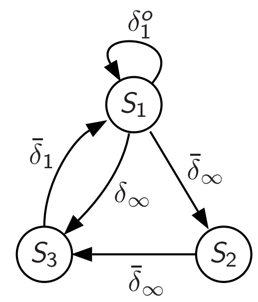


Loop carried output dependence due to `T`

Example 4: Dependence

Program to swap arrays	Dependence Graph

```
for (i=0; i<N; i++)
{
    T = A[i];        /* S1 */
    A[i] = B[i];     /* S2 */
    B[i] = T;        /* S3 */
}
```



Data Dependence Theorem

There exists a dependence from statement S_1 to statement S_2 in common nest of loops if and only if there exist two iteration vectors **i** and **j** for the nest, such that

1. $\mathbf{i} < \mathbf{j}$ or $\mathbf{i} = \mathbf{j}$ and there exists a path from S_1 to S_2 in the body of the loop,
2. statement S_1 accesses memory location M on iteration **i** and statement S_2 accesses location M on iteration **j**, and
3. one of these accesses is a write access.

Anti Dependence and Vectorization

Read precedes Write lexicographically

```
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
   S1: C[i] = A[i+2];
   S2: A[i] = B[i];
}
```

→

```
int A[N], B[N], C[N], i;
for (i=0; i<N; i=i+4) {
   S1: C[i:i+3] = A[i+2:i+5];
   S2: A[i:i+3] = B[i:i+3];
}
```

Anti Dependence and Vectorization

Write precedes Read lexicographically

```
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    S1: A[i] = B[i];
    S2: C[i] = A[i+2];
}
```

```
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    S2: C[i] = A[i+2];
    S1: A[i] = B[i];
}
```

```
int A[N], B[N], C[N], i;
for (i=0; i<N; i=i+4) {
    S2: C[i:i+3] = A[i+2:i+5];
    S1: A[i:i+3] = B[i:i+3];
}
```


True Dependence and Vectorization

Write precedes Read lexicographically

```
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    S1: A[i+2] = C[i];
    S2: B[i] = A[i];
}
```

```
int A[N], B[N], C[N], i;
for (i=0; i<N; i=i+4) {
    S1: A[i+2:i+5] = C[i:i+3];
    S1: B[i:i+3] = A[i:i+3];
}
```


Multiple Dependences and Vectorization

Anti Dependence and True Dependence

```
int A[N], i;
for (i=0; i<N; i++) {
    L1: A[i] = A[i+2];
}
```

```
int A[N], i, temp;
for (i=0; i<N; i++) {
    S1: temp = A[i+2];
    S2: A[i] = temp;
}
```

```
int A[N], T[N], i;
for (i=0; i<N; i++) {
    S1: T[i] = A[i+2];
    S2: A[i] = T[i];
}
```

```
int A[N], T[N], i;
for (i=0; i<N; i=i+4) {
    S1: T[i:i+3] = A[i+2:i+5];
    S2: A[i:i+3] = T[i:i+3];
}
```


Multiple Dependences and Vectorization

True Dependence and Anti Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: A[i] = B[i];
    S2: B[i+2] = A[i+1];
}
```

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S2: B[i+2] = A[i+1];
    S1: A[i] = B[i];
}
```

```
int A[N], B[N], i;
for (i=0; i<N; i=i+4) {
    S2: B[i+2:i+5] = A[i+1:i+4];
    S1: A[i:i+3] = B[i:i+3];
}
```

Observation: Feasibility of Vectorization

- If the source statement lexicographically precedes sink statement in the program, they can be vectorized.

True Dependence and Vectorization

Read precedes Write lexicographically

```
int A[N], i;
for (i=0; i<N; i++) {
    L1: A[i+5] = A[i];
}
```

→

```
int A[N], i, temp;
for (i=0; i<N; i++) {
    S1: temp = A[i];
    S2: A[i+5] = temp;
}
```

↓

```
int A[N], T[N], i;
for (i=0; i<N; i++) {
    S1: T[i] = A[i];
    S2: A[i+5] = T[i];
}
```

←

```
int A[N], T[N], i;
for (i=0; i<N; i=i+4) {
    S1: T[i:i+3] = A[i:i+3];
    S2: A[i+5:i+8] = T[i:i+3];
}
```


Cyclic Dependences and Vectorization

Cyclic True Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i+2] = A[i];
    S2: A[i+1] = B[i];
}
```

Cyclic Anti Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i] = A[i+1];
    S2: A[i] = B[i+2];
}
```

- Rescheduling of statements will not break the cyclic dependence
- The dependence distance from S_2 to $S_1 <$ VF

Cannot Vectorize

Cyclic Dependences and Vectorization

Cyclic True Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i+2] = A[i];
    S2: A[i+5] = B[i];
}
```

Cyclic Anti Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i] = A[i+1];
    S2: A[i] = B[i+5];
}
```

- Rescheduling of statements will not break the cyclic dependence
- The dependence distance from S_2 to $S_1 \geq$ VF

Can Vectorize

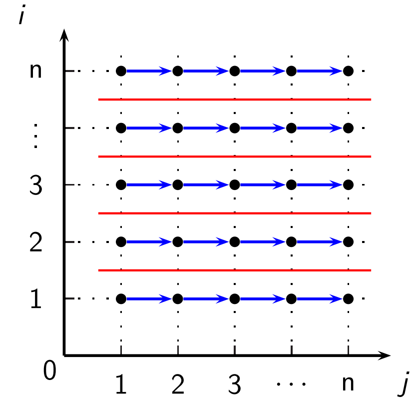
Observation: Feasibility of Vectorization

- If the source statement lexicographically precedes sink statement in the program, they can be vectorized.
- If the dependence distance for all *backward* dependences between two statements is greater than or equal to Vectorization Factor, the statements can be vectorized.

Feasibility of Parallelization

Outer Parallel

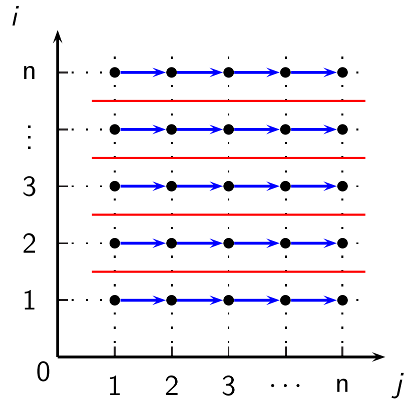
```
for (i=1; i<n; i++)
   for (j=1; j<n; j++)
      A[i][j] = A[i][j+1];
```



Feasibility of Parallelization

Outer Parallel

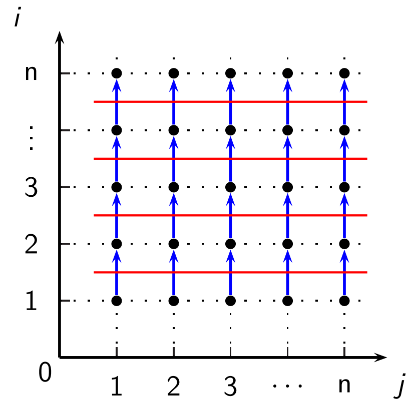
```
for-all (i=1 to n)
   for (j=1; j<n; j++)
      A[i][j] = A[i][j+1];
```



Feasibility of Parallelization

Inner Parallel

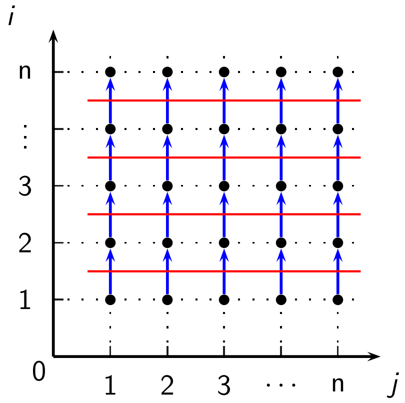
```
for (i=2; i<n; i++)
   for (j=1; j<n; j++)
      A[i][j] = A[i-1][j];
```



Feasibility of Parallelization

Inner Parallel

```
for (i=2; i<n; i++)
   for-all (j=1 to n)
      A[i][j] = A[i-1][j];
```



Part 2

The Lambda Framework

Lambda Framework for Loop Transforms

- Getting loop information (Loop discovery)
- Finding value spaces of induction variables, array subscript functions, and pointer accesses
- Analyzing data dependence
- Performing loop transformations

Loop Transformation Passes in GCC

```
NEXT_PASS (pass_tree_loop);
  {
    struct opt_pass **p = &pass_tree_loop.pass.sub;
    NEXT_PASS (pass_tree_loop_init);
    NEXT_PASS (pass_lim);
    ...
    NEXT_PASS (pass_check_data_deps);
    NEXT_PASS (pass_loop_distribution);
    NEXT_PASS (pass_copy_prop);
    NEXT_PASS (pass_graphite);
      {
        struct opt_pass **p = &pass_graphite.pass.sub;
        NEXT_PASS (pass_graphite_transforms);
        ...
      }
    NEXT_PASS (pass_iv_canon);
    NEXT_PASS (pass_if_conversion);
    NEXT_PASS (pass_vectorize);
      {
        struct opt_pass **p = &pass_vectorize.pass.sub;
        NEXT_PASS (pass_lower_vector_ssa);
        NEXT_PASS (pass_dce_loop);
      }
    NEXT_PASS (pass_predcom);
    NEXT_PASS (pass_complete_unroll);
    NEXT_PASS (pass_slp_vectorize);
    NEXT_PASS (pass_parallelize_loops);
    NEXT_PASS (pass_loop_prefetch);
    NEXT_PASS (pass_iv_optimize);
    NEXT_PASS (pass_tree_loop_done);
  }
```

- Passes on tree-SSA form
  A variant of Gimple IR
- Discover parallelism and transform IR
- Parameterized by some machine dependent features (Vectorization factor, alignment etc.)

Loop Transformation Passes in GCC: Our Focus

Data Dependence	Pass variable name	pass_check_data_deps
	Enabling switch	-fcheck-data-deps
	Dump switch	-fdump-tree-ckdd
	Dump file extension	.ckdd
Loop Distribution	Pass variable name	pass_loop_distribution
	Enabling switch	-ftree-loop-distribution
	Dump switch	-fdump-tree-ldist
	Dump file extension	.ldist
Vectorization	Pass variable name	pass_vectorize
	Enabling switch	-ftree-vectorize
	Dump switch	-fdump-tree-vect
	Dump file extension	.vect
Parallelization	Pass variable name	pass_parallelize_loops
	Enabling switch	-ftree-parallelize-loops=n
	Dump switch	-fdump-tree-parloops
	Dump file extension	.parloops

Compiling for Emitting Dumps

- Other necessary command line switches
  - -O2 -fdump-tree-all
    -O3 enables -ftree-vectorize. Other flags must be enabled explicitly
- Processor related switches to enable transformations apart from analysis
  - -mtune=pentium -msse4
- Other useful options
  - Suffixing -all to all dump switches
  - -S to stop the compilation with assembly generation
  - --verbose-asm to see more detailed assembly dump

Representing Value Spaces of Variables and Expressions

Chain of Recurrences: 3-tuple ⟨Starting Value, modification, stride⟩

```
for (i=3; i<=15; i=i+3)
{
    for (j=11; j>=1; j=j-2)
    {
        A[i+1][2*j-1] = ...
    }
}
```

Entity	CR
Induction variable i	$\{3, +, 3\}$
Induction variable j	$\{11, +, -2\}$
Index expression i+1	$\{4, +, 3\}$
Index expression 2*j-1	$\{21, +, -4\}$

Example 1: Observing Data Dependence

Step 0: Compiling

```
int a[200];
int main()
{
   int i;
   for (i=0; i<150; i++)
   {
      a[i] = a[i+1] + 2;
   }
   return 0;
}
```

gcc -fcheck-data-deps -fdump-tree-ckdd-all -O2 -S datadep.c

Example 1: Observing Data Dependence

Step 1: Examining the control flow graph

Program

```
int a[200];
int main()
{
   int i;
   for (i=0; i<150; i++)
   {
      a[i] = a[i+1] + 2;
   }
   return 0;
}
```

Control Flow Graph

```
<bb 3>:
  # i_13 = PHI <i_3(4), 0(2)>
  i_3 = i_13 + 1;
  D.1955_4 = a[i_3];
  D.1956_5 = D.1955_4 + 2;
  a[i_13] = D.1956_5;
  if (i_3 != 150)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```


Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_3(4), 0(2)>
  i_3 = i_13 + 1;
  D.1955_4 = a[i_3];
  D.1956_5 = D.1955_4 + 2;
  a[i_13] = D.1956_5;
  if (i_3 != 150)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```


Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_3(4), 0(2)>
  i_3 = i_13 + 1;
  D.1955_4 = a[i_3];
  D.1956_5 = D.1955_4 + 2;
  a[i_13] = D.1956_5;
  if (i_3 != 150)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

```
(scalar_evolution = {0, +, 1}_1)
```


Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_3(4), 0(2)>
  i_3 = i_13 + 1;
  D.1955_4 = a[i_3];
  D.1956_5 = D.1955_4 + 2;
  a[i_13] = D.1956_5;
  if (i_3 != 150)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

```
(scalar_evolution = {1, +, 1}_1)
```

Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_3(4), 0(2)>
  i_3 = i_13 + 1;
  D.1955_4 = a[i_3];
  D.1956_5 = D.1955_4 + 2;
  a[i_13] = D.1956_5;
  if (i_3 != 150)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

```
base_address: &a
offset from base address: 0
constant offset from base
                 address: 4
aligned to: 128
(chrec = {1, +, 1}_1)
```


Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_3(4), 0(2)>
  i_3 = i_13 + 1;
  D.1955_4 = a[i_3];
  D.1956_5 = D.1955_4 + 2;
  a[i_13] = D.1956_5;
  if (i_3 != 150)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

```
base_address: &a
offset from base address: 0
constant offset from base
                 address:  0
aligned to: 128
base_object: a[0]
(chrec = {0, +, 1}_1)
```


Example 1: Observing Data Dependence

Step 3: Observing the data dependence information

```
iterations_that_access_an_element_twice_in_A: [1 + 1*x_1]
last_conflict: 149
iterations_that_access_an_element_twice_in_B: [0 + 1*x_1]
last_conflict: 149
Subscript distance: 1

inner loop index: 0
loop nest: (1)
distance_vector: 1
direction_vector: +
```


Example 2: Observing Vectorization and Parallelization

Step 0: Compiling the code with `-O2`

```
int a[256], b[256];
int main()
{
    int i;
    for (i=0; i<256; i++)
    {
        a[i] = b[i];
    }
    return 0;
}
```

- Additional options for parallelization
  `-ftree-parallelize-loops=2 -fdump-tree-parloops-all`
- Additional options for vectorization
  `-fdump-tree-vect-all -msse4 -ftree-vectorize`

Example 2: Observing Vectorization and Parallelization

Step 1: Examining the control flow graph

Program	Control Flow Graph

`int a[256], b[256];`
`int main()`
`{`
`    int i;`
`    for (i=0; i<256; i++)`
`    {`
`        a[i] = b[i];`
`    }`
`    return 0;`
`}`
	`<bb 3>:`
	`  # i_11 = PHI <i_4(4), 0(2)>`
	`  D.2836_3 = b[i_11];`
	`  a[i_11] = D.2836_3;`
	`  i_4 = i_11 + 1;`
	`  if (i_4 != 256)`
	`    goto <bb 4>;`
	`  else`
	`    goto <bb 5>;`
	`<bb 4>:`
	`  goto <bb 3>;`

Example 2: Observing Vectorization and Parallelization

Step 2: Observing the final decision about vectorization

```
parvec.c:5: note: LOOP VECTORIZED.
parvec.c:2: note: vectorized 1 loops in function.
```


Example 2: Observing Vectorization and Parallelization

Step 3: Examining the vectorized control flow graph

Original control flow graph	Transformed control flow graph

`<bb 3>:`
`  # i_11 = PHI <i_4(4), 0(2)>`
`  D.2836_3 = b[i_11];`
`  a[i_11] = D.2836_3;`
`  i_4 = i_11 + 1;`
`  if (i_4 != 256)`
`    goto <bb 4>;`
`  else`
`    goto <bb 5>;`
`<bb 4>:`
`  goto <bb 3>;`
	`<bb 2>:`
	`  vect_pb.7_10 = &b;`
	`  vect_pa.12_15 = &a;`
	`<bb 3>:`
	`  # vect_pb.4_6 = PHI <vect_pb.4_13,`
	`                vect_pb.7_10>`
	`  # vect_pa.9_16 = PHI <vect_pa.9_17,`
	`                vect_pa.12_15>`
	`  vect_var_.8_14 = MEM[vect_pb.4_6];`
	`  MEM[vect_pa.9_16] = vect_var_.8_14;`
	`  vect_pb.4_13 = vect_pb.4_6 + 16;`
	`  vect_pa.9_17 = vect_pa.9_16 + 16;`
	`  ivtmp.13_19 = ivtmp.13_18 + 1;`
	`  if (ivtmp.13_19 < 64)`
	`    goto <bb 4>;`

Example 2: Observing Vectorization and Parallelization

Step 4: Understanding the strategy of parallel execution

- Create threads t_i for $1 \leq i \leq$ MAX_THREADS
- Assigning start and end iteration for each thread
  ⇒ Distribute iteration space across all threads
- Create the following code body for each thread t_i

```
for (j=start_for_thread_i; j<=end_for_thread_i; j++)
{
     /* execute the loop body to be parallelized */
}
```

- All threads are executed in parallel

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1996_6 = __builtin_omp_get_num_threads ();
D.1998_8 = __builtin_omp_get_thread_num ();
D.2000_10 = 255 / D.1997_6;
D.2001_11 = D.2000_10 * D.1997_6;
D.2002_12 = D.2001_11 != 255;
D.2003_13 = D.2002_12 + D.2000_10;
ivtmp.7_14 = D.2003_13 * D.1999_8;
D.2005_15 = ivtmp.7_14 + D.2003_13;
D.2006_16 = MIN_EXPR <D.2005_15, 255>;
if (ivtmp.7_14 >= D.2006_16)
  goto <bb 3>;
```


Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1996_6 = __builtin_omp_get_num_threads ();
D.1998_8 = __builtin_omp_get_thread_num ();
D.2000_10 = 255 / D.1997_6;
D.2001_11 = D.2000_10 * D.1997_6;
D.2002_12 = D.2001_11 != 255;
D.2003_13 = D.2002_12 + D.2000_10;
ivtmp.7_14 = D.2003_13 * D.1999_8;
D.2005_15 = ivtmp.7_14 + D.2003_13;
D.2006_16 = MIN_EXPR <D.2005_15, 255>;
if (ivtmp.7_14 >= D.2006_16)
  goto <bb 3>;
```

Get the number of threads

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1996_6 = __builtin_omp_get_num_threads ();
D.1998_8 = __builtin_omp_get_thread_num ();
D.2000_10 = 255 / D.1997_6;
D.2001_11 = D.2000_10 * D.1997_6;
D.2002_12 = D.2001_11 != 255;
D.2003_13 = D.2002_12 + D.2000_10;
ivtmp.7_14 = D.2003_13 * D.1999_8;
D.2005_15 = ivtmp.7_14 + D.2003_13;
D.2006_16 = MIN_EXPR <D.2005_15, 255>;
if (ivtmp.7_14 >= D.2006_16)
  goto <bb 3>;
```

Get thread identity

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1996_6 = __builtin_omp_get_num_threads ();
D.1998_8 = __builtin_omp_get_thread_num ();
D.2000_10 = 255 / D.1997_6;
D.2001_11 = D.2000_10 * D.1997_6;
D.2002_12 = D.2001_11 != 255;
D.2003_13 = D.2002_12 + D.2000_10;
ivtmp.7_14 = D.2003_13 * D.1999_8;
D.2005_15 = ivtmp.7_14 + D.2003_13;
D.2006_16 = MIN_EXPR <D.2005_15, 255>;
if (ivtmp.7_14 >= D.2006_16)
  goto <bb 3>;
```

Perform load calculations

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1996_6 = __builtin_omp_get_num_threads ();
D.1998_8 = __builtin_omp_get_thread_num ();
D.2000_10 = 255 / D.1997_6;
D.2001_11 = D.2000_10 * D.1997_6;
D.2002_12 = D.2001_11 != 255;
D.2003_13 = D.2002_12 + D.2000_10;
ivtmp.7_14 = D.2003_13 * D.1999_8;
D.2005_15 = ivtmp.7_14 + D.2003_13;
D.2006_16 = MIN_EXPR <D.2005_15, 255>;
if (ivtmp.7_14 >= D.2006_16)
  goto <bb 3>;
```

Assign start iteration to the chosen thread

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1996_6 = __builtin_omp_get_num_threads ();
D.1998_8 = __builtin_omp_get_thread_num ();
D.2000_10 = 255 / D.1997_6;
D.2001_11 = D.2000_10 * D.1997_6;
D.2002_12 = D.2001_11 != 255;
D.2003_13 = D.2002_12 + D.2000_10;
ivtmp.7_14 = D.2003_13 * D.1999_8;
D.2005_15 = ivtmp.7_14 + D.2003_13;
D.2006_16 = MIN_EXPR <D.2005_15, 255>;
if (ivtmp.7_14 >= D.2006_16)
  goto <bb 3>;
```

Assign end iteration to the chosen thread

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1996_6 = __builtin_omp_get_num_threads ();
D.1998_8 = __builtin_omp_get_thread_num ();
D.2000_10 = 255 / D.1997_6;
D.2001_11 = D.2000_10 * D.1997_6;
D.2002_12 = D.2001_11 != 255;
D.2003_13 = D.2002_12 + D.2000_10;
ivtmp.7_14 = D.2003_13 * D.1999_8;
D.2005_15 = ivtmp.7_14 + D.2003_13;
D.2006_16 = MIN_EXPR <D.2005_15, 255>;
if (ivtmp.7_14 >= D.2006_16)
  goto <bb 3>;
```

Start execution of iterations of the chosen thread

Example 2: Observing Vectorization and Parallelization

Step 6: Examining the loop body to be executed by a thread

Control Flow Graph

```
<bb 3>:
  # i_11 = PHI <i_4(4), 0(2)>
  D.1956_3 = b[i_11];
  a[i_11] = D.1956_3;
  i_4 = i_11 + 1;
  if (i_4 != 256)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Parallel loop body

```
<bb 5>:
  i.8_21 = (int) ivtmp.7_18;
  D.2010_23 = *b.10_4[i.8_21];
  *a.11_5[i.8_21] = D.2010_23;
  ivtmp.7_19 = ivtmp.7_18 + 1;
  if (D.2006_16 > ivtmp.7_19)
    goto <bb 5>;
  else
    goto <bb 3>;
```

Example 3: Vectorization but No Parallelization

Step 0: Compiling with
`-O2 -fdump-tree-vect-all -msse4 -ftree-vectorize`

```
int a[624];
int main()
{
    int i;
    for (i=0; i<619; i++)
    {
        a[i] = a[i+4];
    }
    return 0;
}
```


Example 3: Vectorization but No Parallelization

Step 1: Observing the final decision about vectorization

```
vecnopar.c:5: note: LOOP VECTORIZED.
vecnopar.c:2: note: vectorized 1 loops in function.
```


Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph

```
<bb 3>:
  # i_12 = PHI <i_5(4), 0(2)>
  D.2834_3 = i_12 + 4;
  D.2835_4 = a[D.2834_3];
  a[i_12] = D.2835_4;
  i_5 = i_12 + 1;
  if (i_5 != 619)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Vectorized Control Flow Graph

```
<bb 2>:
  vect_pa.10_26 = &a[4];
  vect_pa.15_30 = &a;
<bb 3>:
  # vect_pa.7_27 = PHI <vect_pa.7_28,
                        vect_pa.10_26>
  # vect_pa.12_31 = PHI <vect_pa.12_32,
                        vect_pa.15_30>
  vect_var_.11_29 = MEM[vect_pa.7_27];
  MEM[vect_pa.12_31] = vect_var_.11_29;
  vect_pa.7_28 = vect_pa.7_27 + 16;
  vect_pa.12_32 = vect_pa.12_31 + 16;
  ivtmp.16_34 = ivtmp.16_33 + 1;
  if (ivtmp.16_34 < 154)
    goto <bb 4>;
```


Example 3: Vectorization but No Parallelization

- Step 3: Observing the conclusion about dependence information

```
inner loop index: 0
loop nest: (1 )
distance_vector: 4
direction_vector: +
```

- Step 4: Observing the final decision about parallelization

```
FAILED: data dependencies exist across iterations
```

Example 4: No Vectorization and No Parallelization

Step 0: Compiling the code with -O2

```
int a[256], b[256];
int main ()
{
    int i;
    for (i=0; i<216; i++)
    {
        a[i+2] = b[i] + 5;
        b[i+1] = a[i] + 10;
    }
    return 0;
}
```

- Additional options for parallelization
  -ftree-parallelize-loops=2 -fdump-tree-parloops-all
- Additional options for vectorization
  -fdump-tree-vect-all -msse4 -ftree-vectorize

Example 4: No Vectorization and No Parallelization

- Step 1: Observing the final decision about vectorization

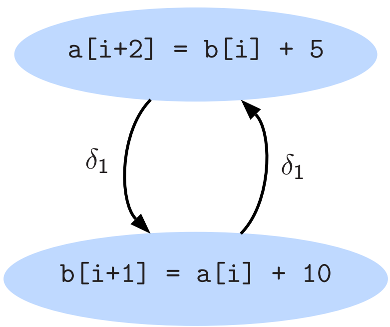
  `noparvec.c:5: note: vectorized 0 loops in function.`

- Step 2: Observing the final decision about parallelization

  `FAILED: data dependencies exist across iterations`

Example 4: No Vectorization and No Parallelization

Step 3: Understanding the dependences that prohibit vectorization and parallelization



Part 3

Transformations Enhancing Vectorization and Parallelization

Transformations Enhancing Vectorization and Parallelization

Some transformations increase the scope of parallelization and vectorization by either enabling them, or by improving their run time performance. Most important of such transformations are:

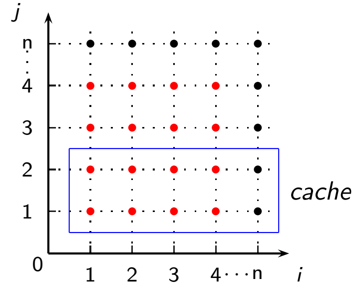
- Loop Interchange
- Loop Distribution
- Loop Fusion
- Peeling

Loop Interchange

Loop Interchange for Vectorization

Original Code

```
for (i=0; i<200; i++) {
   for (j=0; j<200; j++)
      a[j][i] = a[j][i+1];
}
```



- Outer loop is vectorizable
- Mismatch between nesting order of loops and array access

Loop Interchange

Loop Interchange for Vectorization

Original Code

```
for (i=0; i<200; i++) {
   for (j=0; j<200; j++)
      a[j][i] = a[j][i+1];
}
```

After Interchange

```
for (j=0; j<200; j++) {
   for (i=0; i<200; i++)
      a[j][i] = a[j][i+1];
}
```

- Innermost loop is vectorizable
- Loop Interchange improves data locality

Loop Interchange

Loop Interchange for Parallelization

Original Code

```
for (i=1; i<n; i++) {
   for (j=0; j<n; j++)
      A[i][j] = A[i-1][j];
}
```

- Outer Loop - dependence on i, can not be parallelized
- Inner Loop - parallelizable, but synchronization barrier required

  Total number of synchronizations required = n

Loop Interchange

Loop Interchange for Parallelization

Original Code

```
for (i=1; i<n; i++) {
   for (j=0; j<n; j++)
       A[i][j] = A[i-1][j];
}
```

After Interchange

```
for (j=0; j<n; j++) {
   for (i=1; i<n; i++)
       A[i][j] = A[i-1][j];
}
```

- Outer Loop - parallelizable

  Total number of synchronizations required = 1

Loop Distribution

Original Code

```
for (i=0; i<230; i++) {
   S1 : a[i+3] = a[i];
   S2 : b[i] = c[i];
}
```

- True dependence in S_1, no dependence in S_2
- Loop cannot be vectorized or parallelized, but S_2 can be vectorized and parallelized independently

Compile with

```
gcc -O2 -ftree-loop-distribution -fdump-tree-ldist
```


Loop Distribution

Control Flow Graph

```
<bb 3>:
  # i_13 = PHI <i_6(4), 0(2)>
  D.2692_3 = i_13 + 3;
  D.2693_4 = a[i_13];
  a[D.2692_3] = D.2693_4;
  D.2694_5 = c[i_13];
  b[i_13] = D.2694_5;
  i_6 = i_13 + 1;
  if (i_6 != 230)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Distributed Control Flow Graph

```
<bb 6>:
  # i_11 = PHI <i_18(7), 0(2)>
  D.2692_12 = i_11 + 3;
  D.2693_7 = a[i_11];
  a[D.2692_12] = D.2693_7;
  i_18 = i_11 + 1;
  if (i_18 != 230)
    goto <bb 6>;
<bb 8>:
  # i_13 = PHI <i_6(4), 0(8)>
  D.2694_5 = c[i_13];
  b[i_13] = D.2694_5;
  i_6 = i_13 + 1;
  if (i_6 != 230)
    goto <bb 8>;
```


Loop Distribution

After Distribution

```
for (i=0; i<230; i++)
    S1 : a[i+3] = a[i];
for (i=0; i<230; i++)
    S2 : b[i] = c[i];
```

- S_2 can now be independently parallelized or vectorized
- S_1 runs sequentially

Loop Fusion for Locality

Original Code

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        a[i][j] = b[i];
for (k=0; k<n; k++)
    for (l=0; l<n; l++)
        b[k] = a[k][l];
```

- Large reuse distance for array a and b, high chances of cache miss
- If loops i and k are parallelized, 2 synchronizations required
- Outer loops i and k can be fused
- Fusing inner loops j and l will introduce a spurious backward dependence on b

Loop Fusion for Locality

Original Code

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        a[i][j] = b[i];
for (k=0; k<n; k++)
    for (l=0; l<n; l++)
        b[k] = a[k][l];
```

Fused Code

```
for (i=0; i<n; i++) {
    for (j=0; j<n; j++)
        a[i][j] = b[i];
    for (l=0; l<n; l++)
        b[i] = a[i][l];
}
```

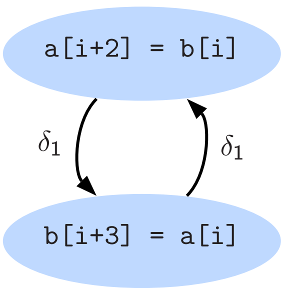
- Reduced reuse distance for array a and b, low chances of cache miss
- If outer loop i is parallelized, only 1 synchronization required

Peeling

Peeling for Vectorization

Original Code

```
for (i=0; i<n; i++)
{
    S1: a[i+2] = b[i];
    S2: b[i+3] = a[i];
}
```



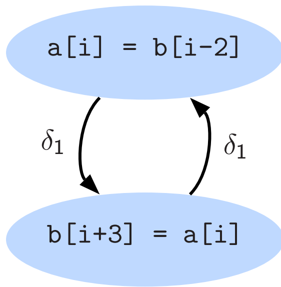
- Cyclic Dependence, dependence distance for *backward* dependence $= 3 < \text{VF}$
- Cannot vectorize

Peeling

Peeling for Vectorization

Transformed Code

```
for (i=0; i<2; i++)
    S2: b[i+3] = a[i];
for (i=2; i<n-2; i++) {
    S1: a[i] = b[i-2];
    S2: b[i+3] = a[i];
}
```



- Cyclic Dependence, dependence distance for *backward* dependence $= 5 > \text{VF}$
- Can vectorize

Peeling

Peeling for Parallelization

Original Code

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

- dependence on i, can not be parallelized

Total number of synchronizations required = n

Peeling

Peeling for Parallelization

Original Code

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

Transformed Code

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

- Outer Loop parallelizable

Total number of synchronizations required = 1

Part 4

Advanced Issues in Vectorization and Parallelization

Advanced Issues in Vectorization and Parallelization

- What code can be vectorized?
- How to force the alignment of data accesses for
  - compile time misalignment
  - run time misalignment
- How to handle undetermined aliases?
- When is vectorization profitable?
- When is parallelization profitable?

Understanding the cost model of vectorizer and parallelizer

Unvectorizable Loops

```
int *a, *b;
int main() {
    while (*a != NULL)
    {
         *a++ = *b--;
    }
}
```

novec.c:6: note: not vectorized: number of iterations cannot be computed.

Reducing Compile Time Misalignment by Peeling

```
int a[256], b[256];
int main ()
{
     int i;
     for (i=0; i<203; i++)
          a[i+2] = b[i+2];
}
```

```
peel.c:5: note: misalign = 8 bytes of ref b[D.2836_4]
peel.c:5: note: misalign = 8 bytes of ref a[D.2836_4]
```


Reducing Compile Time Misalignment by Peeling

Observing the final decision about alignment

```
peel.c:5: note: Try peeling by 2
peel.c:5: note: Alignment of access forced using peeling.
peel.c:5: note: Peeling for alignment will be applied.

peel.c:5: note: known peeling = 2.
peel.c:5: note: niters for prologue loop: 2
peel.c:5: note: Cost model analysis:
     prologue iterations: 2
     epilogue iterations: 1
```


Reducing Compile Time Misalignment by Peeling

An aligned vectorized code can consist of three parts

- Peeled Prologue - Scalar code for alignment
- Vectorized body - Iterations that are vectorized
- Epilogue - Residual scalar iterations

Reducing Compile Time Misalignment by Peeling

Control Flow Graph

```
<bb 3>:
  # i_12 = PHI <i_6(4), 0(2)>
  D.2690_4 = i_12 + 2;
  D.2691_5 = b[D.2690_4];
  a[D.2690_4] = D.2691_5;
  i_6 = i_12 + 1;
  if (i_6 != 203)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Vectorized Control Flow Graph

```
<bb 3>:
  # ivtmp.8_27 = PHI <ivtmp.8_28(4),
                              0(2)>
  D.2908_16 = i_7 + 2;
  D.2909_17 = b[D.2908_16];
  a[D.2908_16] = D.2909_17;
  ivtmp.8_28 = ivtmp.8_27 + 1;
  if (ivtmp.8_28 < 2)
    goto <bb 3>;
  else
    goto <bb 5>;
```

2 Iterations of Prologue

Reducing Compile Time Misalignment by Peeling

Control Flow Graph

```
<bb 3>:
  # i_12 = PHI <i_6(4), 0(2)>
  D.2690_4 = i_12 + 2;
  D.2691_5 = b[D.2690_4];
  a[D.2690_4] = D.2691_5;
  i_6 = i_12 + 1;
  if (i_6 != 203)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Vectorized Control Flow Graph

```
<bb 5>:
  vect_pb.15_4 = &b[4];
  vect_pa.20_8 = &a[4];
<bb 6>:
  # vect_pb.12_5 = PHI <vect_pb.12_6,
                        vect_pb.15_4>
  # vect_pa.17_9 = PHI <vect_pa.17_3,
                        vect_pa.20_8>
  vect_var_.16_7 = MEM[vect_pb.12_5];
  MEM[vect_pa.17_9] = vect_var_.16_7;
  vect_pb.12_6 = vect_pb.12_5 + 16;
  vect_pa.17_3 = vect_pa.17_9 + 16;
  ivtmp.21_52 = ivtmp.21_51 + 1;
  if (ivtmp.21_52 < 50)
    goto <bb 10>;
```

200 Iterations of Vector Code

Reducing Compile Time Misalignment by Peeling

Control Flow Graph

```
<bb 3>:
  # i_12 = PHI <i_6(4), 0(2)>
  D.2690_4 = i_12 + 2;
  D.2691_5 = b[D.2690_4];
  a[D.2690_4] = D.2691_5;
  i_6 = i_12 + 1;
  if (i_6 != 203)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Vectorized Control Flow Graph

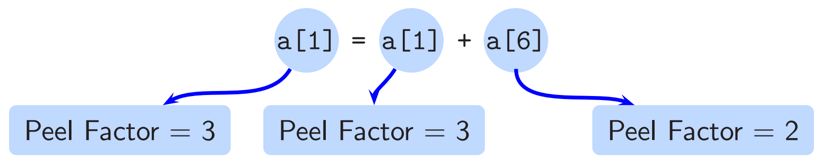
```
<bb 7>:
  tmp.10_42 = ivtmp.8_28 + 200;
<bb 8>:
  # i_29 = PHI <i_35(9), tmp.10_42(7)>
  # ivtmp.3_31 = PHI <ivtmp.3_36(9),
                      tmp.11_43(7)>
  D.2908_32 = i_29 + 2;
  D.2909_33 = b[D.2908_32];
  a[D.2908_32] = D.2909_33;
  i_35 = i_29 + 1;
  ivtmp.3_36 = ivtmp.3_31 - 1;
  if (ivtmp.3_36 != 0)
    goto <bb 8>;
```

1 Iteration of Epilogue

Cost Model for Peeling

```
int a[256];
int main ()
{
     int i;
     for (i=4; i<253; i++)
         a[i-3] = a[i-3] + a[i+2];

}
```



Cost Model for Peeling

```
int a[256];
int main ()
{
     int i;
     for (i=4; i<253; i++)
         a[i-3] = a[i-3] + a[i+2];

}
```

```
a[1] = a[1] + a[6]
```

Maximize alignment with minimal peel factor

Cost Model for Peeling


```
int a[256];
int main ()
{
     int i;
     for (i=4; i<253; i++)
         a[i-3] = a[i-3] + a[i+2];

}
```


Peel the loop by 3

Reducing Run Time Misalignment by Versioning

```
int a[256], b[256];
int main (int x, int y)
{
     int i;
     for (i=0; i<200; i++)
         a[i+y] = b[i+x];
}
```

```
version.c:5: note: Unknown alignment for access: b
version.c:5: note: Unknown alignment for access: a
```


Reducing Run Time Misalignment by Versioning

```
D.2921_16 = (long unsigned int) x_5(D);
base_off.6_17 = D.2921_16 * 4;
vect_pb.7_18 = &b + base_off.6_17;
D.2924_19 = (long unsigned int) vect_pb.7_18;
D.2925_20 = D.2924_19 & 15;
D.2926_21 = D.2925_20 >> 2;
D.2927_22 = -D.2926_21;
D.2928_23 = (unsigned int) D.2927_22;
prolog_loop_niters.8_24 = D.2928_23 & 3;
D.2932_37 = prolog_loop_niters.8_24 == 0;
if (D.2932_37 != 0)
    goto <bb 6>;
else
    goto <bb 3>;
```

Compute address misalignment as 'addr & (vectype_size -1)'

Reducing Run Time Misalignment by Versioning

```
D.2921_16 = (long unsigned int) x_5(D);
base_off.6_17 = D.2921_16 * 4;
vect_pb.7_18 = &b + base_off.6_17;
D.2924_19 = (long unsigned int) vect_pb.7_18;
D.2925_20 = D.2924_19 & 15;
D.2926_21 = D.2925_20 >> 2;
D.2927_22 = -D.2926_21;
D.2928_23 = (unsigned int) D.2927_22;
prolog_loop_niters.8_24 = D.2928_23 & 3;
D.2932_37 = prolog_loop_niters.8_24 == 0;
if (D.2932_37 != 0)
    goto <bb 6>;
else
    goto <bb 3>;
```

Compute number of prologue iterations

Reducing Run Time Misalignment by Versioning

```
D.2921_16 = (long unsigned int) x_5(D);
base_off.6_17 = D.2921_16 * 4;
vect_pb.7_18 = &b + base_off.6_17;
D.2924_19 = (long unsigned int) vect_pb.7_18;
D.2925_20 = D.2924_19 & 15;
D.2926_21 = D.2925_20 >> 2;
D.2927_22 = -D.2926_21;
D.2928_23 = (unsigned int) D.2927_22;
prolog_loop_niters.8_24 = D.2928_23 & 3;
D.2932_37 = prolog_loop_niters.8_24 == 0;
if (D.2932_37 != 0)
    goto <bb 6>;
else
    goto <bb 3>;
```

If accesses can be aligned, go to vectorized code

Reducing Run Time Misalignment by Versioning

```
D.2921_16 = (long unsigned int) x_5(D);
base_off.6_17 = D.2921_16 * 4;
vect_pb.7_18 = &b + base_off.6_17;
D.2924_19 = (long unsigned int) vect_pb.7_18;
D.2925_20 = D.2924_19 & 15;
D.2926_21 = D.2925_20 >> 2;
D.2927_22 = -D.2926_21;
D.2928_23 = (unsigned int) D.2927_22;
prolog_loop_niters.8_24 = D.2928_23 & 3;
D.2932_37 = prolog_loop_niters.8_24 == 0;
if (D.2932_37 != 0)
    goto <bb 6>;
else
    goto <bb 3>;
```

Else go to sequential code

Versioning for Undetermined Aliases

```
int a[256];
int main (int *b)
{
    int i;
    for (i=0; i<200; i++)
        *b++ = a[i];
}
```

```
version.c:5: note: misalign = 0 bytes of ref a[i_15]
version.c:5: note: can't force alignment of ref: *b_14
version.c:5: note: versioning for alias required: can't
determine dependence between a[i_15] and *b_14
version.c:5: note: create runtime check for data references
a[i_15] and *b_14
```

Versioning for Undetermined Aliases

Control Flow Graph

```
<bb 3>:
  # b_14 = PHI <b_6, b_4(D)>
  # i_15 = PHI <i_7(4), 0(2)>
  D.2907_5 = a[i_15];
  *b_14 = D.2907_5;
  b_6 = b_14 + 4;
  i_7 = i_15 + 1;
  if (i_7 != 200)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Vectorized Control Flow Graph

```
<bb 2>:
  vect_pa.6_12 = &a;
  vect_p.9_11 = b_4(D);
  D.2919_13 = vect_pa.6_12 + 16;
  D.2920_8 = D.2919_13 < vect_p.9_11;
  D.2921_17 = vect_p.9_11 + 16;
  D.2922_18 = D.2921_17 < vect_pa.6_12;
  D.2923_19 = D.2920_8 || D.2922_18;
  if (D.2923_19 != 0)
    goto <bb 3>;
  else
    goto <bb 6>;
```

Check for dependence within VF

Versioning for Undetermined Aliases

Control Flow Graph

```
<bb 3>:
  # b_14 = PHI <b_6, b_4(D)>
  # i_15 = PHI <i_7(4), 0(2)>
  D.2907_5 = a[i_15];
  *b_14 = D.2907_5;
  b_6 = b_14 + 4;
  i_7 = i_15 + 1;
  if (i_7 != 200)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Vectorized Control Flow Graph

```
<bb 3>:
  #vect_pa.10_30 = PHI <vect_pa.10_31,
                        vect_pa.13_29>
  #vect_p.15_34 = PHI <vect_p.15_35,
                        vect_p.18_33>
  #ivtmp.19_36 = PHI <ivtmp.19_37, 0>
  vect_var_.14_32 = MEM[vect_pa.10_30];
  MEM[vect_p.15_34] = vect_var_.14_32;
  vect_pa.10_31 = vect_pa.10_30 + 16;
  vect_p.15_35 = vect_p.15_34 + 16;
  ivtmp.19_37 = ivtmp.19_36 + 1;
  if (ivtmp.19_37 < 50)
    goto <bb 3>;
  else
    goto <bb 9>;
```

Execute vector code if no aliases within VF

Versioning for Undetermined Aliases

Control Flow Graph

```
<bb 3>:
  # b_14 = PHI <b_6, b_4(D)>
  # i_15 = PHI <i_7(4), 0(2)>
  D.2907_5 = a[i_15];
  *b_14 = D.2907_5;
  b_6 = b_14 + 4;
  i_7 = i_15 + 1;
  if (i_7 != 200)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Vectorized Control Flow Graph

```
<bb 6>:
  #b_20 = PHI <b_4(D)(6), b_26(8)>
  #i_21 = PHI <0(6), i_27(8)>
  #ivtmp.3_23 = PHI <200, ivtmp.3_28>
  D.2907_24 = a[i_21];
  *b_20 = D.2907_24;
  b_26 = b_20 + 4;
  i_27 = i_21 + 1;
  ivtmp.3_28 = ivtmp.3_23 - 1;
  if (ivtmp.3_28 != 0)
    goto <bb 6>;
  else
    goto <bb 9>;
```

Execute scalar code if aliases are within VF

Profitability of Vectorization

```
int a[256], b[256];
int main ()
{
     int i;
     for (i=0; i<50; i++)
         a[i] = b[i*4];
}
```

```
vec.c:5: note: cost model: the vector iteration cost = 10
divided by the scalar iteration cost = 2 is greater or
equal to the vectorization factor = 4.

vec.c:5: note: not vectorized: vectorization not profitable.
```

Profitability of Vectorization

```
short int a[256], b[256];
int main ()
{
     int i;
     for (i=0; i<50; i++)
         a[i] = b[i*4];
}
```

Vectorization Factor = 8

VF x scalar iteration cost > vector iteration cost

```
vec.c:5: note: LOOP VECTORIZED.
vec.c:2: note: vectorized 1 loops in function.
```


Cost Model of Vectorizer

Vectorization is profitable when

$$SIC * niters + SOC > VIC * \left(\frac{niters - PL\_ITERS - EP\_ITERS}{VF}\right) + VOC$$

SIC = scalar iteration cost
VIC = vector iteration cost
VOC = vector outside cost
VF = vectorization factor
PL_ITERS = prologue iterations
EP_ITERS = epilogue iterations
SOC = scalar outside cost

Cost Model of Vectorizer

```
int main (int *a, int *b)
{
     int i, n;
     for (i=0; i<n; i++)
     *a++ = *b--;
}
```

```
vec.c:4: note: versioning for alias required: can't
       determine dependence between *b_19 and *a_18
vec.c:4: note: Cost model analysis:
  Vector inside of loop cost: 4
  Vector outside of loop cost: 14
  Scalar iteration cost: 2
  Scalar outside cost: 1
  prologue iterations: 0
  epilogue iterations: 2
  Calculated minimum iters for profitability: 12
```


Cost Model of Vectorizer

```
int main (int * restrict a, int * restrict b)
{
    int i, n;
     for (i=0; i<n; i++)
         *a++ = *b--;
}
```

```
vec.c:4: note: Cost model analysis:
  Vector inside of loop cost: 3
  Vector outside of loop cost: 16
  Scalar iteration cost: 2
  Scalar outside cost: 7
  prologue iterations: 2
  epilogue iterations: 2
  Calculated minimum iters for profitability: 5
```

Cost Model of Parallelizer

```
int a[500];
int main ()
{
     int i;
     for (i=0; i<350; i++)
         a[i] = a[i] + 2;
}
```

Compile with:

`gcc -O2 -fdump-tree-parloops -ftree-parallelize-loops=4`

Loop not parallelized as number of iterations per thread ≤ 100

Cost Model of Parallelizer

```
int a[500];
int main ()
{
     int i;
     for (i=0; i<350; i++)
         a[i] = a[i] + 2;
}
```

Compile with:

`gcc -O2 -fdump-tree-parloops -ftree-parallelize-loops=3`

SUCCESS: may be parallelized

Cost Model of Parallelizer

Inner Parallelism

```
int i, j;
for (i=0; i<450; i++)
   for (j=0; j<420; j++)
       a[i][j] = a[i-1][j];
```

Compile with:

`gcc -O2 -fdump-tree-parloops -ftree-parallelize-loops=4`

```
distance_vector:   1   0
direction_vector:    +    =
FAILED: data dependencies exist across iterations
```


Cost Model of Parallelizer

Outer Parallelism

```
int i, j;
for (j=0; j<420; j++)
   for (i=0; i<450; i++)
       a[i][j] = a[i-1][j];
```

Compile with:

`gcc -O2 -fdump-tree-parloops -ftree-parallelize-loops=4`

```
distance_vector:   0   1
direction_vector:    =    +
SUCCESS: may be parallelized
```

Cost Model of Parallelizer

```
D.2000_5 = __builtin_omp_get_num_threads ();
D.2001_6 = (unsigned int) D.2000_5;
D.2002_7 = __builtin_omp_get_thread_num ();
D.2003_8 = (unsigned int) D.2002_7;
D.2004_9 = 419 / D.2001_6;
D.2005_10 = D.2004_9 * D.2001_6;
D.2006_11 = D.2005_10 != 419;
D.2007_12 = D.2006_11 + D.2004_9;
ivtmp.7_13 = D.2007_12 * D.2003_8;
D.2009_14 = ivtmp.7_13 + D.2007_12;
D.2010_15 = MIN_EXPR <D.2009_14, 419>;
if (ivtmp.7_13 >= D.2010_15)
  goto <bb 3>;
```


Cost Model of Parallelizer

```
D.2000_5 = __builtin_omp_get_num_threads ();
D.2001_6 = (unsigned int) D.2000_5;
D.2002_7 = __builtin_omp_get_thread_num ();
D.2003_8 = (unsigned int) D.2002_7;
D.2004_9 = 419 / D.2001_6;
D.2005_10 = D.2004_9 * D.2001_6;
D.2006_11 = D.2005_10 != 419;
D.2007_12 = D.2006_11 + D.2004_9;
ivtmp.7_13 = D.2007_12 * D.2003_8;
D.2009_14 = ivtmp.7_13 + D.2007_12;
D.2010_15 = MIN_EXPR <D.2009_14, 419>;
if (ivtmp.7_13 >= D.2010_15)
  goto <bb 3>;
```

Get the number of threads

Cost Model of Parallelizer

```
D.2000_5 = __builtin_omp_get_num_threads ();
D.2001_6 = (unsigned int) D.2000_5;
D.2002_7 = __builtin_omp_get_thread_num ();
D.2003_8 = (unsigned int) D.2002_7;
D.2004_9 = 419 / D.2001_6;
D.2005_10 = D.2004_9 * D.2001_6;
D.2006_11 = D.2005_10 != 419;
D.2007_12 = D.2006_11 + D.2004_9;
ivtmp.7_13 = D.2007_12 * D.2003_8;
D.2009_14 = ivtmp.7_13 + D.2007_12;
D.2010_15 = MIN_EXPR <D.2009_14, 419>;
if (ivtmp.7_13 >= D.2010_15)
  goto <bb 3>;
```

Get thread identity

Cost Model of Parallelizer

```
D.2000_5 = __builtin_omp_get_num_threads ();
D.2001_6 = (unsigned int) D.2000_5;
D.2002_7 = __builtin_omp_get_thread_num ();
D.2003_8 = (unsigned int) D.2002_7;
D.2004_9 = 419 / D.2001_6;
D.2005_10 = D.2004_9 * D.2001_6;
D.2006_11 = D.2005_10 != 419;
D.2007_12 = D.2006_11 + D.2004_9;
ivtmp.7_13 = D.2007_12 * D.2003_8;
D.2009_14 = ivtmp.7_13 + D.2007_12;
D.2010_15 = MIN_EXPR <D.2009_14, 419>;
if (ivtmp.7_13 >= D.2010_15)
  goto <bb 3>;
```

Perform load calculations

Cost Model of Parallelizer

```
D.2000_5 = __builtin_omp_get_num_threads ();
D.2001_6 = (unsigned int) D.2000_5;
D.2002_7 = __builtin_omp_get_thread_num ();
D.2003_8 = (unsigned int) D.2002_7;
D.2004_9 = 419 / D.2001_6;
D.2005_10 = D.2004_9 * D.2001_6;
D.2006_11 = D.2005_10 != 419;
D.2007_12 = D.2006_11 + D.2004_9;
ivtmp.7_13 = D.2007_12 * D.2003_8;
D.2009_14 = ivtmp.7_13 + D.2007_12;
D.2010_15 = MIN_EXPR <D.2009_14, 419>;
if (ivtmp.7_13 >= D.2010_15)
  goto <bb 3>;
```

Assign start iteration to the chosen thread

Cost Model of Parallelizer

```
D.2000_5 = __builtin_omp_get_num_threads ();
D.2001_6 = (unsigned int) D.2000_5;
D.2002_7 = __builtin_omp_get_thread_num ();
D.2003_8 = (unsigned int) D.2002_7;
D.2004_9 = 419 / D.2001_6;
D.2005_10 = D.2004_9 * D.2001_6;
D.2006_11 = D.2005_10 != 419;
D.2007_12 = D.2006_11 + D.2004_9;
ivtmp.7_13 = D.2007_12 * D.2003_8;
D.2009_14 = ivtmp.7_13 + D.2007_12;
D.2010_15 = MIN_EXPR <D.2009_14, 419>;
if (ivtmp.7_13 >= D.2010_15)
  goto <bb 3>;
```

Assign end iteration to the chosen thread

Cost Model of Parallelizer

```
D.2000_5 = __builtin_omp_get_num_threads ();
D.2001_6 = (unsigned int) D.2000_5;
D.2002_7 = __builtin_omp_get_thread_num ();
D.2003_8 = (unsigned int) D.2002_7;
D.2004_9 = 419 / D.2001_6;
D.2005_10 = D.2004_9 * D.2001_6;
D.2006_11 = D.2005_10 != 419;
D.2007_12 = D.2006_11 + D.2004_9;
ivtmp.7_13 = D.2007_12 * D.2003_8;
D.2009_14 = ivtmp.7_13 + D.2007_12;
D.2010_15 = MIN_EXPR <D.2009_14, 419>;
if (ivtmp.7_13 >= D.2010_15)
  goto <bb 3>;
```

Start execution of iterations of the chosen thread

Parallelization and Vectorization in GCC : Conclusions

- Chain of recurrences seems to be a useful generalization
- Interaction between different passes is not clear due to fixed order
- Auto-vectorization and auto-parallelization can be improved by enhancing the dependence analysis framework
- Efficient cost models are needed to automate legal transformation composition