A Crash Course in Compilers for Parallel Computing

Mary Hall

Fall, 2008

Outline of Lecture

I. Introduction to course
- Relevance to compiler researchers, parallel programmers, architects

II. What is Data Dependence?
- Define for scalar variables
- Extend to arrays and loop
- Brief discussion of pointer-based computations and objects

III. Loop iteration reordering
- For exploiting parallelism
- For managing locality

Suggested Reading

- Intent of course is to be self-contained (no outside reading needed)
- Some material taken from:
  - Optimizing Compilers for Modern Architectures by Randy Allen and Ken Kennedy, Morgan Kaufmann Publishers, 2002.

I. Introduction and Motivation

- Foundation for compiler research …
- … But there are other opportunities
- Parallel Programmers:
  - Learn to reason about correct and efficient parallel programs
- Architects:
  - Capabilities and limitations of compilers

I. Introduction and Motivation, Why now?

- Technology trends have led us to the "multi-core paradigm shift"
  - Can no longer rely on performance from increasing clock rates
- Parallel programming will become dominant
  - To achieve performance
  - To enhance software capability

Overview of "Crash Course"

- L1: Data Dependence Analysis and Parallelization
- L2: Loop Reordering Transformations
- L3: Locality Analysis and Optimization
- L4: Autotuning Compiler Technology

II. Data Dependence

- **Definition:**
  Two memory accesses are involved in a data dependence if they may refer to the same memory location and one of the references is a write.

  A data dependence can either be between two distinct program statements or two different dynamic executions of the same program statement.

- Two important uses of data dependence information (among others):
  Parallelization: no data dependence between two computations ➔ parallel execution safe
  Locality optimization: absence of data dependences & presence of reuse ➔ reorder memory accesses for better data locality

More generally, How are Data Dependences Used?

- Scalar analog: data-flow analysis
  - Partial redundancy elimination, loop-invariant code motion, ...
- Identify parallel loops *(data parallel)*
- Identify independent tasks *(task parallel)*
- Manage communication and locking
  - Possible dependence ➔ protect with lock
  - Proven dependence ➔ communicate to use
- Determine safety of reordering transformations and data structure transformations
  - For parallelism and locality
- Exploiting instruction-level parallelism
  - E.g., Software pipelining
- Hints to hardware
  - Prefetch, coherence, ...

Data Dependence of Scalar Variables

True (flow) dependence

```
a     =
      = a
```

Anti-dependence

```
      = a
a     =
```

Output dependence

```
a     =
a     =
```

Input dependence (for locality)

```
      = a
      = a
```

Definition: Data dependence exists from a reference instance i to i' iff

- either i or i' is a write operation
- i and i' refer to the same variable
- i executes before i'

Some Definitions (from Allen & Kennedy)

- **Definition 2.5:**
  - Two computations are equivalent if, on the same inputs,
    - they produce identical outputs
    - the outputs are executed in the same order
- **Definition 2.6:**
  - A reordering transformation
    - changes the order of statement execution
    - without adding or deleting any statement executions.
- **Definition 2.7:**
  - A reordering transformation preserves a dependence if
    - it preserves the relative execution order of the dependences' source and sink.

Fundamental Theorem of Dependence

- **Theorem 2.2:**
  - Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.
- Now we will discuss abstractions and algorithms to determine whether reordering transformations preserve dependences…

In Today's Lecture: Parallelizable Loops

Forall or Doall loops:

Loops whose iterations can execute in parallel (a particular reordering transformation)

Example

```
forall (i=1; i<=n; i++)
    A[i] = B[i] + C[i];
```

Meaning?

Each iteration can execute independently of others

Free to schedule iterations in any order

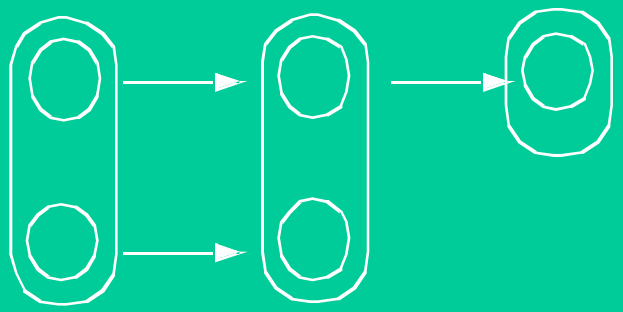
Why are parallelizable loops an important concept?

Source of scalable, balanced work

Common to scientific, multimedia, graphics & other domains

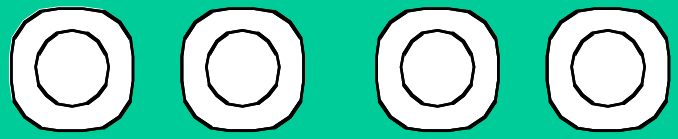
Data Dependence for Arrays

```
for (i=2; i<5; i++)
     A[i] = A[i-2]+1;
```



Loop-Carried dependence

```
for (i=1; i<=3; i++)
     A[i] = A[i]+1;
```



Loop-Independent dependence

- Recognizing parallel loops (intuitively)
  - Find data dependences in loop
  - No dependences crossing iteration boundary ➔ parallelization of loop's iterations is safe

0. Pre-pass: Loop Normalization

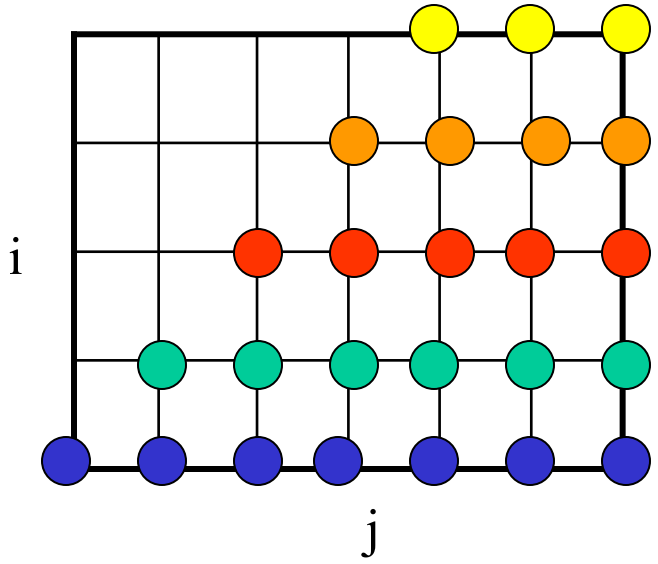
- We will talk about memory accesses with respect to the iteration spaces of loop nests.
- Assumes loop iteration counts begin at "1" and step by "1"
- Loops can always be ***normalized*** to ensure this property:

```
for (i=4; i<=12; i+=2)
    A[i] = …
```

```
for (i=1; i<=5; i++)
    A[i*2 + 2] = ...
```

1. Characterize Iteration Space

```
for (i=1;i<=5; i++)
  for (j=i;j<=7; j++)
      ...
```



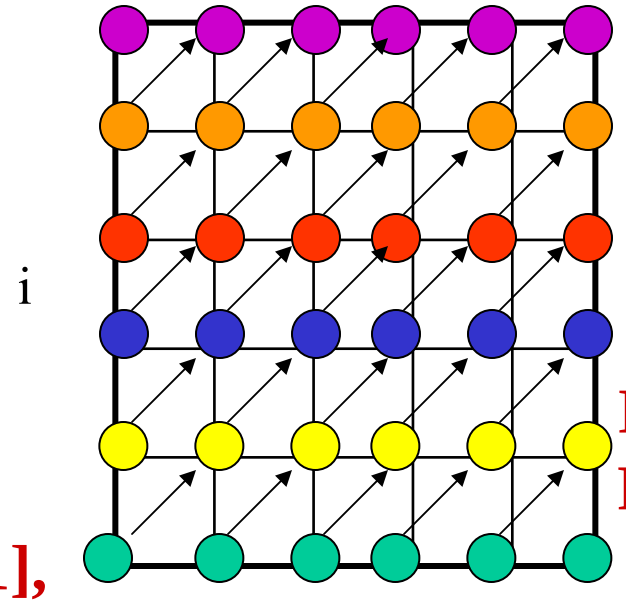
$1 \le i \le 5$

$i \le j \le 7$

- **Iteration instance:** represented as coordinates in iteration space
  - *n*-dimensional discrete cartesian space for *n* deep loop nests
- **Lexicographic order:** Sequential execution order of iterations
  [1,1], [1,2], …, [1,6],[1,7],
  [2,2], [2,3], …, [2,6], …
- Iteration I (a vector) is lexicographically less than I', I<I' , iff
  there exists c (i_1, …, i_{c-1}) = (i'_1, …, i'_{c-1}) and $i_c < i'_c$.

2. Compare Memory Accesses across Dynamic Instances in Iteration Space

```
N = 6;
for (i=1; i<N; i++)
    for (j=1; j<N; j++)
        A[i+1,j+1] = A[i,j] * 2.0;
```



I=[1,1], Write A[2,2]

I'=[2,2], Read A[2,2]

How to describe relationship between two dynamic instances?

e.g., **I=[1,1]** and **I'=[2,2]**

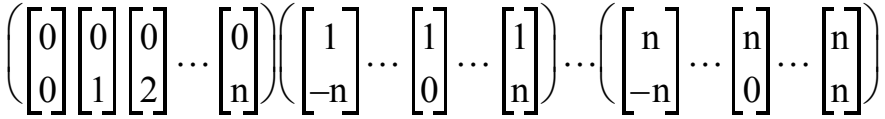
Distance Vectors

```
N = 6;
for (i=1; i<N; i++)
    for (j=1; j<N; j++)
        A[i+1,j+1] = A[i,j] * 2.0;
```

- Distance vector = **[1,1]**
- A loop has a distance vector D if there exists data dependence from iteration vector **I** to a later vector **I'**, and **D = I' - I**.
- Since I' > I, **D >= 0**.
  (D is lexicographically greater than or equal to 0).

Distance and Direction Vectors

- Distance vectors: (infinitely large set)



- Direction vectors: (realizable if 0 or lexicographically positive)

  ([=,=],[=,<],[<,>], [<,=], [<.<])

- Common notation:

0	=
+	<
-	>
+/-	*

Parallelization Test: 1-Dimensional Loop

- **Examples:**

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

```
for (j=1; j<N; j++)
    B[j] = B[j-1] + 1;
```

- **Dependence (Distance and Direction) Vectors?**

- **Test for parallelization:**

  - A loop is parallelizable if for all data dependences D e **D**, D = 0

n-Dimensional Loop Nests

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

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

- **Distance and direction vectors?**
- **Definition:**
  $D = (d_1, \ldots d_n)$ is loop-carried at level i if d_i is the first nonzero element.

Test for Parallelization

The *i* th loop of an *n*-dimensional loop is parallelizable if there does not exist any level *i* data dependences.

The *i* th loop is parallelizable if for all dependences $D = (d_1, \ldots d_n)$,

either

$(d_1, \ldots d_{i-1}) > 0$

or

$(d_1, \ldots d_i) = 0$

Parallelization Algorithm

- **For each pair of dynamic accesses to the same array within a loop nest:**
  - determine if there exists a dependence between that pair
- **Key points:**
  - n^2 tests for n accesses in loop!
  - a single access is compared with itself
  - includes accesses in all loops within a nest

Dependence Testing

- **Question so far:**
  - What is the distance/direction (in the iteration space) between two dynamic accesses to the same memory location?
- **Simpler question:**
  - Can two data accesses ever refer to the same memory location?

```
for (i=11;i<=20;i++)
  A[i] = A[I-1]+ 3;
```

```
for (i=11;i<=20;i++)
   A[I] = A[I-10]+ 1;
```

Restrict to an Affine Domain

```
for (i=1; i<N; i++)
   for (j=1; j<N j++) {
      A[i+2*j+3, 4*i+2*j, 3*i] = ...;
      ... = A[1, 2*i+1, j];
}
```

- Only use loop bounds and array indices which are integer linear functions of loop variables.
- **Non-affine example:**

```
 for (i=1; i<N; i++)
    for (j=1; j<N j++) {
       A[i*j] = A[i*(j-1)];
       A[B[i]] = A[B[j]];
 }
```

Equivalence to Integer Programming

- Need to determine if F(i) = G(i'), where i and i' are iteration vectors, with constraints i,i' >= L, U>= i, i'
- **Example:**

```
for (i=2; i<=100; i++)
    A[i] = A[i-1];
```

- **Inequalities:**

0 <= i1 <= 100, i2 = i1 - 1, i2 <= 100

integer vector I, AI <= b

- **Integer Programing is NP-complete**
  - O(size of the coefficients)
  - $O(n^n)$

Example Using SUIF Calculator

```
SUBROUTINE MATTRN(A, N)
 INTEGER N
 REAL A(N,N)
 INTEGER NM1, J, I
 REAL TEMP
 NM1 = N - 1
 IF (NM1 .EQ. 0) RETURN
 DO 200 J = 1, NM1
     DO 100 I = J+1, N
       TEMP = A(I,J)
       A(I,J) = A(J,I)
       A(J,I) = TEMP
 100 CONTINUE
 200 CONTINUE
 ...
```

```
iterr = [   1 <= jr
            jr <= nm1
            jr+1 <= ir
            ir <= n
                        ]
iterw = [   1 <= jw
            jw <= nm1
            jw+1 <= iw
            iw <= n
                         ]
access = [  ir = jw
            jr = iw
                         ]
dependence =
      {iterr,iterw,access}
dependence.intsol()
```

Such Formulations Often Overkill and Typically Used Only When Needed

- Consider the following simple, exact test
- Definition:
  - SIV= Single Induction Variable
- Strong SIV:
  - An SIV subscript for loop index I is strong if it has the form <aI + c1, aI′ + c2>
- Dependence distance can be calculated exactly as follows:
  - d = I′ - I = (c1 - c2) / a

Summary of Lecture

- Data dependence is a fundamental concept in compilers for high-performance computing.
- Data dependence can be used to determine the safety of reordering transformations
  - preserving dependences = preserving "meaning"
- Iteration, distance and direction vectors are abstractions for understanding whether reordering transformations preserve dependences.
- Dependence testing on array accesses in loops has been shown to be equivalent to integer programming.

Next Week:

- How do we use dependence information to decide safety of reordering transformations other than parallelization?