COMP 515: Advanced Compilation for Vector and Parallel Processors

Prof. Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP515

Homework #3 Problem Statement

1. **Solve exercise 5.6 in book**
   - Your solution should be legal for all values of K (note that the value of K is invariant in loop I)

Exercise 5.6: What vector code should be generated for the following loop?

```
DO I = 1, 100
    A(I) = B(K) + C(I)
    B(I+1) = A(I) + D(I)
END DO
```

Homework 3 Solution

- **In general, the following loop**

```
DO I = 1, 100
    A(I) = B(K) + C(I)
    B(I+1) = A(I) + D(I)
END DO
```

can be split into two loops as follows (index set splitting):

```
S0: B(…) = …
 DO I = 1, min(K-1,100)    ! All reads of B(K) return value from S0 (before entry to loop)
 S1:    A(I) = B(K) + C(I)    ! Only dep on B is loop independent anti dep, when I=K-1
 S2: B(I+1) = A(I) + D(I)   ! Only dependence on A is loop independent flow
 END DO
 DO I = max(K,1), 100            ! All reads of B(K) return value from S2 when I=K-1
 S3:    A(I) = B(K) + C(I)    ! No dependence on B
 S4:    B(I+1) = A(I) + D(I) ! Only dependence on A is loop independent flow
 END DO
```

- **This transformation is correct for all values of K**

Homework 3 Solution (contd)

- **The vectorization then becomes straightforward for the two loops:**

```
! All reads of B(K) return value from S0 (before entry to loop)
S1:   A(1:min(K-1,100)) = B(K) + C(1:min(K-1,100))
S2:   B(2:min(K,101)) = A(1:min(K-1,100)) + D(1:min(K-1,100))
! All reads of B(K) return value from S2 when I=K-1
S3:   A(max(K,1):100) = B(K) + C(max(K,1):100)
S4:   B(max(K+1,2):101) = A(max(K,1):100) + D(max(K,1):100)
```

- **Finally, check the solutions for different cases**
- **Case 1: K <= 1**
  - **Statements S1 and S2 become no-ops**
  - **Statements S3 and S4 run with vector lengths = 100 exactly**
- **Case 2: K >= 101**
  - **Statements S1 and S2 run with vector lengths = 100 exactly**
  - **Statements S3 and S4 become no-ops**
- **Case 3: 2 <= K <= 100**
  - **Statements S1 and S2 run with vector lengths = K-1 exactly**
  - **Statements S3 and S4 run with vector lengths = 100-K+1 exactly**

Midterm Review of Chapters 1-5

(Chapter 6 is in scope for Exam 1, but excluded from this review since it was covered recently in class)

Compiler Challenges for High Performance Architectures

Allen and Kennedy, Chapter 1

Bernstein's Conditions [1966]

- When is it safe to run two tasks R1 and R2 in parallel?
  - If none of the following holds:
    1. R1 writes into a memory location that R2 reads
    2. R2 writes into a memory location that R1 reads
    3. Both R1 and R2 write to the same memory location
- How can we convert this to loop parallelism?
  - Think of loop iterations as tasks
- Does this apply to sequential loops embedded in an explicitly parallel program?
  - Impact of memory model on ordering of read operations

Dependence: Theory and Practice

Allen and Kennedy, Chapter 2

Dependences

- Formally:

  There is a data dependence from statement S_1 to statement S_2 (S_2 depends on S_1) if:

  1. Both statements access the same memory location and at least one of them stores onto it, and
  2. There is a feasible run-time execution path from S_1 to S_2

Load Store Classification

- Quick review of dependences classified in terms of load-store order:
  1. True dependences (RAW hazard)
     - S_2 depends on S_1 is denoted by $S_1 \delta S_2$
  2. Antidependence (WAR hazard)
     - S_2 depends on S_1 is denoted by $S_1 \delta^{-1} S_2$
  3. Output dependence (WAW hazard)
     - S_2 depends on S_1 is denoted by $S_1 \delta^0 S_2$

Formal Definition of Loop Dependence

- Theorem 2.1 Loop Dependence:
  There exists a dependence from statements S_1 to statement S_2 in a common nest of loops if and only if there exist two iteration vectors i and j for the nest, such that
  (1) i < j or i = j and there is 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.

- Follows from the definition of dependence

Reordering Transformations

- A reordering transformation is any program transformation that merely changes the order of execution of the code, without adding or deleting any executions of any statement
- A reordering transformation does not eliminate dependences
- A reordering transformation preserves a dependence if it preserves the relative execution order of the source and sink of that dependence.
- Fundamental Theorem of Dependence:
  - Any reordering transformation that preserves every dependence in a program preserves the meaning of that program
  - Proof by contradiction. Theorem 2.2 in the book.
- A transformation is said to be valid or legal for the program to which it applies if it preserves all dependences in the program.

Distance & Direction Vectors

- Consider a dependence in a loop nest of n loops
  - Statement S_1 on iteration i is the source of the dependence
  - Statement S_2 on iteration j is the sink of the dependence

- The distance vector is a vector of length n, d(i,j) such that: $d(i,j)_k = j_k - i_k$
  - We normalize distance vectors for loops in which the index step size is not equal to 1 (but usually prefer to normalize the loops to step of +1 instead)

- The direction vector is a vector of length n, r(i,j) such that: $r(i,j)_k = j_k \; op_k \; i_k$, where op_k is a relational operator (<, >, =, <=, >=, !=, *)

Implausible Distance & Direction Vectors

- A distance vector is implausible if its leftmost nonzero element is negative i.e., if the vector is lexicographically less than the zero vector
- Likewise, a direction vector is implausible if its leftmost non "=" component is not "<"
- No dependence in a sequential program can have an implausible distance or direction vector as this would imply that the sink of the dependence occurs before the source.

Direction Vector Transformation

- Theorem 2.3. Direction Vector Transformation. Let T be a transformation that is applied to a loop nest and that does not rearrange the statements in the body of the loop. Then the transformation is valid if, after it is applied, none of the direction vectors for dependences with source and sink in the nest has a leftmost non- "=" component that is ">" i.e., none of the transformed direction vectors become implausible.

- Follows from Fundamental Theorem of Dependence:
  - All dependences exist
  - None of the dependences have been reversed

Loop-carried and Loop-independent Dependences

- If in a loop statement S_2 depends on S_1, then there are two possible ways of this dependence occurring:

1. S_1 and S_2 execute on different iterations
   - This is called a loop-carried dependence.
2. S_1 and S_2 execute on the same iteration
   - This is called a loop-independent dependence.

Simple Vectorization Algorithm

procedure vectorize (L, D)

// L is the maximal loop nest containing the statement.

// D is the dependence graph for statements in L.

find the set $\{S_1, S_2, \ldots, S_m\}$ of maximal strongly-connected regions in the dependence graph D restricted to L (Tarjan);

construct L_p from L by reducing each S_i to a single node and compute D_p, the dependence graph naturally induced on L_p by D;

let $\{p_1, p_2, \ldots, p_m\}$ be the m nodes of L_p numbered in an order consistent with D_p (use topological sort);

```
for i = 1 to m do begin
        if p_i is a dependence cycle then
    generate a DO-loop nest around the statements in p_i;
        else
    directly rewrite p_i in Fortran 90, vectorizing it with respect to every loop
      containing it;
 end
end vectorize
```

Problems With Simple Vectorization

```
      DO I = 1, N
                DO J = 1, M
S1                      A(I+1,J) = A(I,J) + B
                ENDDO
      ENDDO
```

- Dependence from S_1 to itself with d(i, j) = (1,0)
- Key observation: Since dependence is at level 1, we can manipulate the other loop!
- Can be converted to:

```
      DO I = 1, N
S1       A(I+1,1:M) = A(I,1:M) + B
      ENDDO
```

- The simple algorithm does not capitalize on such opportunities

Advanced Vectorization Algorithm

procedure codegen(R, k, D);
// R is the region for which we must generate code.
// k is the minimum nesting level of possible parallel loops.
// D is the dependence graph among statements in R..
find the set $\{S_1, S_2, \ldots, S_m\}$ of maximal strongly-connected regions in the dependence graph D
    restricted to R;
construct R_p from R by reducing each S_i to a single node and
compute D_p, the dependence graph naturally induced on R_p by D;
let $\{p_1, p_2, \ldots, p_m\}$ be the m nodes of R_p numbered in an order
    consistent with D_p (use topological sort to do the numbering);
for i = 1 to m do begin
    if p_i is cyclic then begin
        generate a level-k DO statement;
        let D_i be the dependence graph consisting of all dependence edges in D that are at level
            k+1 or greater and are internal to p_i;
        codegen (p_i, k+1, D_i);
        generate the level-k ENDDO statement;
    end
    else
        generate a vector statement for p_i in $r(p_i)$-k+1 dimensions, where r (p_i) is the number of
            loops containing p_i;
end

Dependence Testing

Allen and Kennedy, Chapter 3

The General Problem

```
DO i1 = L1, U1
  DO i2 = L2, U2
            …
    DO in = Ln, Un
 S1                  A(f1(i1,…,in),…,fm(i1,…,in)) = …
 S2                  … = A(g1(i1,…,in),…,gm(i1,…,in))
    ENDDO
    ...
  ENDDO
ENDDO
```

Under what conditions is the following true for iterations α *and* β *?*

$$f_i(\alpha) = g_i(\beta) \text{ for all } i, 1 \le i \le m$$

Note that the number of equations equals the rank of the array,

and the number of variables is twice the number of loops that enclose both array references (two iteration vectors)

Basics: Complexity

A subscript equation is said to be

- ZIV if it contains no index (zero index variable)
- SIV if it contains only one index (single index variable)
- MIV if it contains more than one index (multiple index variables)

For Example:

```
A(5,I+1,j) = A(1,I,k) + C
      First subscript equation is ZIV
      Second subscript equation is SIV
      Third subscript equation is MIV
```

Dependence Testing: Overview

- Partition subscripts of a pair of array references into separable and coupled groups
- Classify each subscript as ZIV, SIV or MIV
- For each separable subscript apply single subscript test. If not done goto next step
- For each coupled group apply multiple subscript test
- If still not done, merge all direction vectors computed in the previous steps into a single set of direction vectors

Linear Diophantine Equations

- A basic result tells us that there are values for $x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n$ so that

$$a_1x_1 - b_1y_1 + \ldots + a_nx_n - b_ny_n = \gcd(a_1, \ldots, a_n, b_1, \ldots, b_n)$$

  What's more, $\gcd(a_1, \ldots, a_n, b_1, \ldots, b_n)$ is the smallest number this is true for.

- As a result, the equation has a solution iff $\gcd(a_1, \ldots, a_n, b_1, \ldots, b_n)$ divides $b_0 - a_0$
  - But the solution may not be in the region (loop iteration values) of interest
- Exercise: try this result on the A(4*i+2) & A(4*i+4) example

Real Solutions

- Unfortunately, the gcd test is less useful then it might seem.
- Useful technique is to show that the equation has no solutions in region of interest ==> explore real solutions for this purpose
- Solving h(x) = 0 is essentially an integer programming problem. Linear programming techniques are used as an approximation.
- Since the function is continuous, the Intermediate Value Theorem says that a solution exists iff:

$$\min_R h \le 0 \le \max_R h$$

Banerjee Inequality

- We need an easy way to calculate $\min_R h$ and $\max_R h$.

- Definitions:

$$h_i^+ = \max_{Ri} h(x_i, y_i) \qquad a^+ = \begin{cases} a & a \geq 0 \\ 0 & a < 0 \end{cases}$$

$$h_i^- = \min_{Ri} h(x_i, y_i) \qquad a^- = \begin{cases} |a| & a < 0 \\ 0 & a \geq 0 \end{cases}$$

- a^+ and a^- are both >= 0 and are called the positive part and negative part of a, so that $a = a^+ - a^-$

Banerjee Inequality

- **Theorem 3.3 (Banerjee). Let D be a direction vector, and h be a dependence function. h = 0 can be solved in the region R iff:**

$$\sum_{i=1}^{n} H_i^-(D_i) \le b_0 - a_0 \le \sum_{i=1}^{n} H_i^+(D_i)$$

Proof: Immediate from Lemma 3.3 and the IMV.

Preliminary Transformations

Chapter 4 of Allen and Kennedy

Loop Normalization

- Transform loop so that
  - The new stride becomes +1 (more important)
  - The new lower bound becomes +1 (less important)
- To make dependence testing as simple as possible
- Serves as information gathering phase

Enhancing Fine-Grained Parallelism

Chapter 5 of Allen and Kennedy

Chapter 2's Codegen

- Codegen: tries to find parallelism using transformations of loop distribution and statement reordering
- If we deal with loops containing cyclic dependences early on in the loop nest, we can potentially vectorize more loops

- Goal in Chapter 5: To explore other transformations to exploit parallelism

Loop Interchange: Safety

- Theorem 5.1 **Let D(i,j) be a direction vector for a dependence in a perfect nest of loops. Then the direction vector for the same dependence after a permutation of the loops in the nest is determined by applying the same permutation to the elements of D(i,j).**

```
DO I = 1, L
  DO J = 1, M
    DO K = 1, N
      A(I+1,J+1,K) = B(I,J,K)
    ENDDO
  ENDDO
ENDDO
```

Dependence: (<, <, =)

```
DO I = 1, L
  DO K = 1, N
    DO J = 1, M
      A(I+1,J+1,K) = B(I,J,K)
    ENDDO
  ENDDO
ENDDO
```

Dependence: (<, =, <)

Loop Interchange: Safety

- Theorem 5.2 **A permutation of the loops in a perfect nest is legal if and only if the direction matrix, after the same permutation is applied to its columns, has no ">" direction as the leftmost non-"=" direction in any row.**
- Follows from Theorem 5.1 and Theorem 2.3

Example:

$$\begin{array}{ccc} i & j & k \end{array} \qquad\qquad \begin{array}{ccc} j & k & i \end{array}$$

$$\begin{pmatrix} < & < & = \\ < & = & > \end{pmatrix} \longrightarrow \begin{pmatrix} < & = & < \\ = & \boxed{>} & < \end{pmatrix}$$

Scalar Expansion and its use in Removing Anti and Output Dependences

```
     DO I = 1, N
S1     T = A(I)
S2     A(I) = B(I)
S3     B(I) = T
     ENDDO
```

- **Scalar Expansion:**

```
     DO I = 1, N
S1     T$(I) = A(I)
S2     A(I) = B(I)
S3     B(I) = T$(I)
     ENDDO
     T = T$(N)
```

- **leads to:**

```
S1      T$(1:N) = A(1:N)
S2      A(1:N) = B(1:N)
S3      B(1:N) = T$(1:N)
        T = T$(N)
```

Scalar Renaming

- Renaming algorithm partitions all definitions and uses into equivalent classes, each of which can occupy different memory locations.
- Use the definition-use graph to:
  - Pick definition
  - Add all uses that the definition reaches to the equivalence class
  - Add all definitions that reach any of the uses…
  - ..until fixed point is reached
- Example:

```
     IF (…) THEN
S1       T = …
     ELSE
S2       T = …
     ENDIF
S3   … = T
S4     T = …
S5   … = T
```

```
IF (…) THEN
     T1 = …
ELSE
     T1 = …
ENDIF
… = T1
T2 = …
… = T2
```

Array Renaming

```
DO I = 1, N
S1      A(I) = A(I-1) + X
S2      Y(I) = A(I) + Z
S3      A(I) = B(I) + C
  ENDDO
```

- $S_1 \, \delta_\infty \, S_2$ $S_2 \, \delta_\infty^{-1} \, S_3$ $S_3 \, \delta_1 \, S_1$ $S_1 \, \delta_\infty^{0} \, S_3$

- **Rename A(I) to A'(I):**

```
   DO I = 1, N
S1      A'(I) = A(I-1) + X
S2      Y(I) = A'(I) + Z
S3      A(I) = B(I) + C
   ENDDO
```

- **Dependences remaining:** $S_1 \, \delta_\infty \, S_2$ **and** $S_3 \, \delta_1 \, S_1$

Node Splitting

- **Sometimes Renaming fails**

```
DO I = 1, N
  S1:       A(I) = X(I+1) + X(I)
  S2:       X(I+1) = B(I) + 32
ENDDO
```

- **Recurrence kept intact by renaming algorithm**

Node Splitting

```
DO I = 1, N
S1:  A(I) = X(I+1) + X(I)
S2:  X(I+1) = B(I) + 32
ENDDO
```

- Break critical antidependence
- Make copy of read from which antidependence emanates

```
DO I = 1, N
S1': X$(I) = X(I+1)
S1: A(I) = X$(I) + X(I)
S2: X(I+1) = B(I) + 32
ENDDO
```

- Recurrence broken
- Vectorized to

```
S1':  X$(1:N) = X(2:N+1)
S2:   X(2:N+1) = B(1:N) + 32
S1:   A(1:N) = X$(1:N) + X(1:N)
```

Index-set Splitting

- **Subdivide loop into different iteration ranges to achieve partial parallelization**
  - **Threshold Analysis** [Strong SIV, Weak Crossing SIV]
  - **Loop Peeling** [Weak Zero SIV]
  - **Section Based Splitting** [Variation of loop peeling]

Threshold Analysis

```
DO I = 1, 20
     A(I+20) = A(I) + B
ENDDO
```

Vectorize to..

```
A(21:40) = A(1:20) + B
```

```
DO I = 1, 100
     A(I+20) = A(I) + B
ENDDO
```

Strip mine to..

```
DO I = 1, 100, 20
   DO i = I, I+19
     A(i+20) = A(i) + B
   ENDDO
ENDDO
```

Vectorize this (the `DO I = 1, 100, 20` / `DO i = I, I+19` loops)

Loop Peeling

- **Source of dependence is a single iteration**

```
DO I = 1, N
    A(I) = A(I) + A(1)
ENDDO
```

Loop peeled to..

```
A(1) = A(1) + A(1)
DO I = 2, N
    A(I) = A(I) + A(1)
ENDDO
```

Vectorize to..

```
A(1) = A(1) + A(1)
A(2:N)= A(2:N) + A(1)
```

Run-time Symbolic Resolution

- "Breaking Conditions"

```
DO I = 1, N
   A(I+L) = A(I) + B(I)
ENDDO
Transformed to..
IF(L.LE.0 .OR. L.GT.N) THEN
   A(L+1:N+L) = A(1:N) + B(1:N)
ELSE
   DO I = 1, N
      A(I+L) = A(I) + B(I)
   ENDDO
ENDIF
```

Loop Skewing

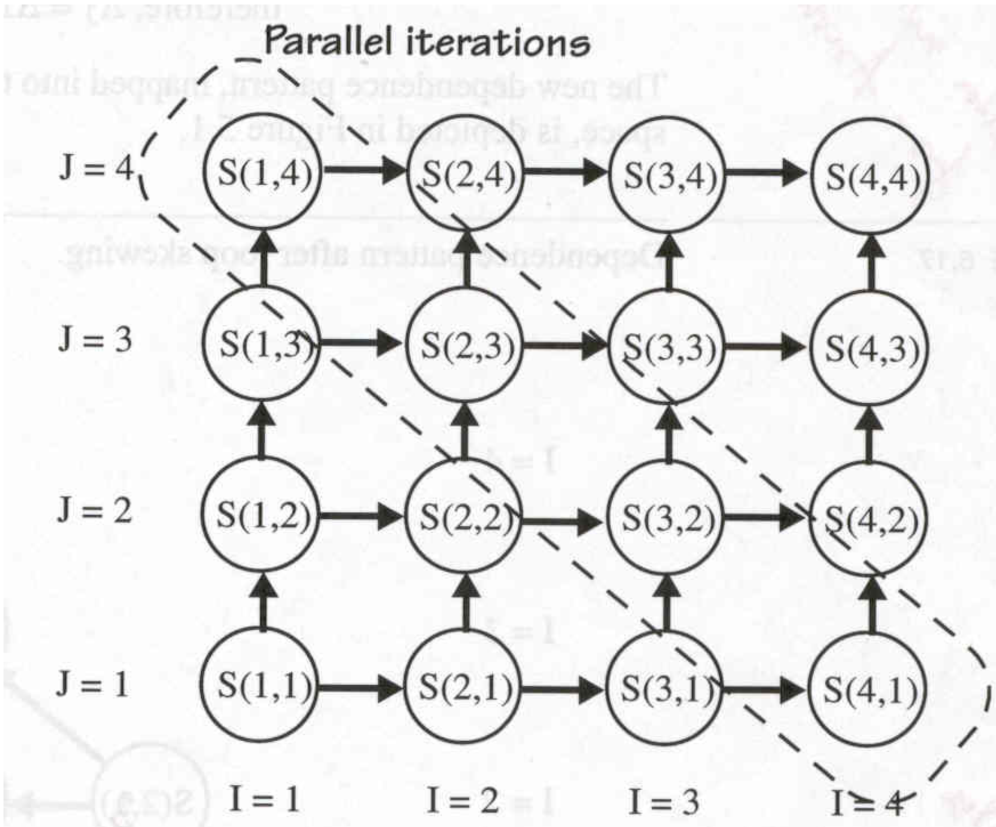
- **Reshape Iteration Space to uncover parallelism**

```
DO I = 1, N
  DO J = 1, N
             (=,<)
S: A(I,J) = A(I-1,J) + A(I,J-1)
    (<,=)
  ENDDO
ENDDO
```

Parallelism not apparent

Loop Skewing

- **Dependence Pattern before loop skewing**



Loop Skewing

- **Do the following transformation called loop skewing**

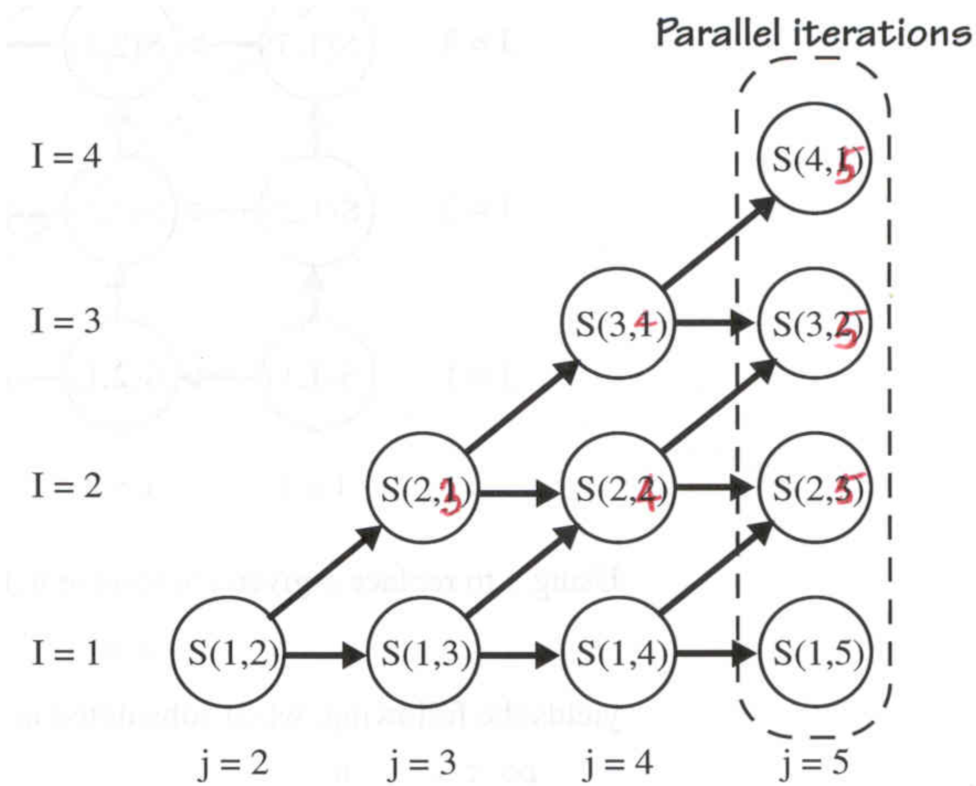
```
jj=J+I or J=jj-I

DO I = 1, N
   DO jj = I+1, I+N
    J = jj - I              (=,<)
S:  A(I,J) = A(I-1,J) + A(I,J-1)
      (<,<)
   ENDDO
ENDDO

Note: Direction Vector Changes, but statement body remains the same
(Examples in textbook usually copy propagate J=jj-I in all uses of J)
```

Loop Skewing

- **Dependence pattern after loop skewing**



Midterm exam reminder (Exam 1)

- **Take-home exam (3 hours)**
  - **Open book: textbook only, no other resources**
  - **Made available today (Thursday, Oct 15th), and needs to be returned to Annepha Pemberton in Duncan Hall room 3080 by Oct 22nd**
  - **Scope of exam is Chapters 1-6 of textbook**