Module 13: INTRODUCTION TO COMPILERS FOR HIGH PERFORMANCE COMPUTERS
Lecture 25: Supercomputing Applications

The Lecture Contains:

Previous Next

Similarly

```
For I = 1 to 10 do
    For J = 1 to 10 do
        A(I,J) = 0;
For I = 1 to 10 do
    A(I,I) = 1;
```

Can be replaced by

```
For I = 1 to 10 do
    For J = 1 to 10 do
        A(I,J) = 0;
    A(I,I) = 1;
```

Loop Unswitching

A loop may be split into two loops. For example the following loop

```
For I = 1 to 100 do
    if Bexpr
        then
            X[I] = A[I] + B[I]
        else
            X[I] = A[I] - B[I]
    Endif
Endfor
```

May be replaced by

```
If Bexpr
    then
        For I = 1 to 100 do
            X[I] = A[I] + B[I]
        Endfor
    else
        For I = 1 to 100 do
            X[I] = A[I] - B[I]
        Endfor
Endif
```

INTRODUCTION TO COMPILERS FOR HIGH PERFORMANCE COMPUTERS

Supercomputing Applications

- Used in scientific computing requiring very large compute time
- Weather prediction: For a days prediction at 109 fpo/sec requires 3 hours
- Used in biology, genetic engineering, astrophysics, aerospace, nuclear and particle physics, medicine, tomography ...

Power of supercomputers comes from

- Hardware technology: Faster machines
- Multilevel architectural parallelism
  Vector handles arrays with a single instruction
  Parallel lot of processors each capable of executing an independent instruction stream
  VLIW handles many instructions in a single cycle

Programming Paradigms

Vector Machines very close to sequential machines. Different in use of arrays
Parallel Machines deal with a system of processes. Has individual threads execution. Synchronization is a very important issue.

- Important things to remeber are
  - Aavoid deadlocks
  - Prevent race conditions
  - Avoid too many parallel threads
- Performance evaluation criterion
  - Speed up vs number of processors
  - Synchronization over heads
  - Number of processors which can be kept busy

- Current software technology is unable to handle all these issues
- Book keeping is still done by the users
- Main directions of research
  - Design of concurrent algorithms
  - Design of concurrent languages
  - Construction of tools to do software development
    - Sequential compilers and book keeping tools
    - Parallelizing and vectoring compilers
    - Message passing libraries
  - Development of mathematical libraries
- Languages Fortran, C, Java, X10 etc.
- Dusty decks problem
  - Conversion of large body of existing sequential programs developed over last 40 years
  - Several billions lines of code and manual conversion is not possible
  - Restructuring compilers are required

Important Problems

Restructuring

- Identify parts of program to take advantage of characteristics of machine
- Manual coding is not possible
- Compilers are more efficient

Scheduling

- Exploit parallelism on a given machine
- Static scheduling: done at compile time
- Dynamic scheduling: done at run time
  - High overhead
  - Implemented through low level calls without involving OS (auto scheduling compilers)
- Auto scheduling compilers
  - Offer new environments for executing parallel programs
  - Small overhead
  - Parallel execution of fine grain granularity is possible
- Loop scheduling: Singly nested vs multiple nested
- Runtime overheads: Scheduling, synchronization, inter processor communication

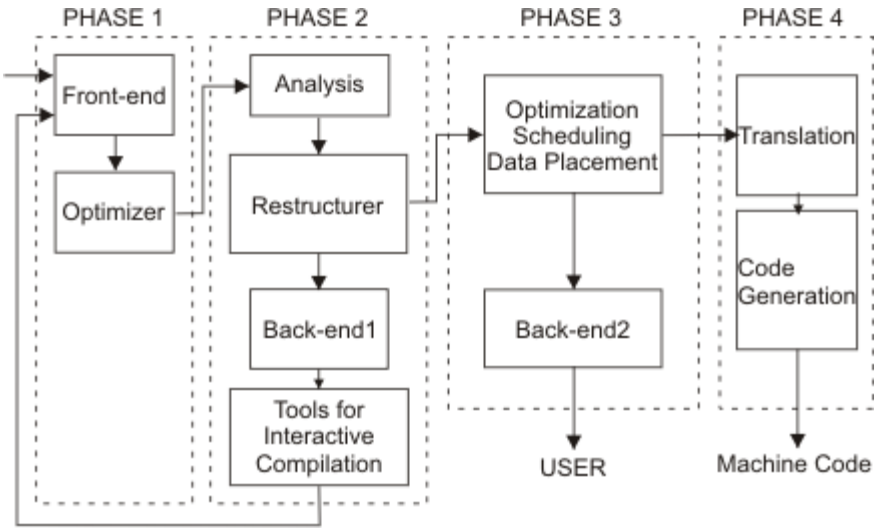
Previous Next

Module 13: INTRODUCTION TO COMPILERS FOR HIGH PERFORMANCE COMPUTERS
Lecture 25: Supercomputing Applications

Sources and Types of Parallelism

- Structured: Identical tasks on different data sets
- Unstructured: Different data streams and different instructions
- Algorithm level: Appropriate algorithms and data structures
- Programming:
  - Specify parallelism in parallel languages
  - Write sequential code and use compilers
  - Use course grain parallelism: independent modules
  - Use medium grain parallelism: loop level
  - Use fine grain parallelism: basic block or statement
- Expressing parallelism in programs
  - No good languages
  - Programmers are unable to exploit whatever little is available

Model of Compiler



CODE OPTIMIZATION

High Level Analysis for optimization

- Common subexpression elimination
- Copy propagation
- Dead code elimination
- Code motion
- Strength reduction
- Constant folding
- Loop unrolling
- Induction variable simplification

Previous Next

Module 13: INTRODUCTION TO COMPILERS FOR HIGH PERFORMANCE COMPUTERS
Lecture 25: Supercomputing Applications

Data Dependence Analysis

- Check whether program can be restructured
- GCD test
- Banerjee's Test
- I-test
- Omega test
- Architectural specific tests

Program Restructurer

- Loop restructuring
- Statement reordering
- Loop (distribution) fission
- Breaking dependence cycle: Node splitting
- Loop interchanging
- Loop skewing
- Loop jamming
- Handling while loops

Technique to Improve Detection of Parallelism: Interactive Compilation

- Back-end to user
- Compiler directives
  - Force Parallel
  - No Parallel
  - Max-trips count
  - Task identification
  - No side effects
- Incremental Analysis

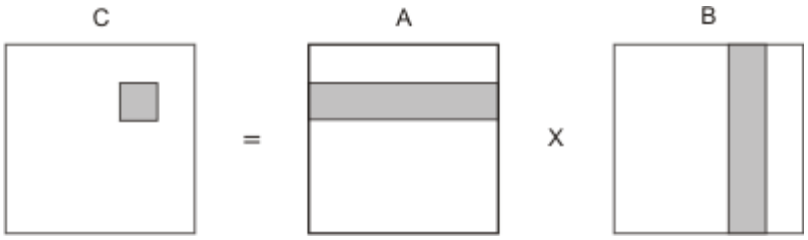
Scalar Processors

- Machines use standard CPUs
- Main memory is in hierarchy: RAM and cache
- Most programs exhibit locality of reference
  - Same items are referred very close in time (temporal)
  - Nearby locations are referred frequently (spatial)
- Programmers do not have control over memory
- Programmers can take advantage of cache

Previous Next

Matrix Multiplication

```
for i := 1 to n do
for j := 1 to n do
for k := 1 to n do
c[i,j] = c[i,j] + a[i,k] * b[k,j]
endfor
endfor
endfor
```



Code for Scalar Processor

```
loop:           Code for Scalar Processor
                ;f1 holds value of C[i,j]
                ;r1 holds value of n
                ;r2 holds address of A[i,1]
                ;r3 holds address of B[1,j]
                ;r13 holds size of row of B
                ;4 is the size of an element of A
                ;loop label
loadf f2, (r2)  ;load A[i,k]
loadf f3, (r3)  ;load B[k,j]
mpyf f4, f2, f3 ;A[i,k] * B[k,j]
addf f1, f1, f4 ;C = C + A * B
addi r2, r2, #4 ;incr pointer to A[i,k+1]
add r3, r3, r13 ;incr pointer to B[k+1,j]
subi r1, r1, #1 ;decr r1 by 1
bnz r1, loop    ;branch to loop if r1 6= 0
```

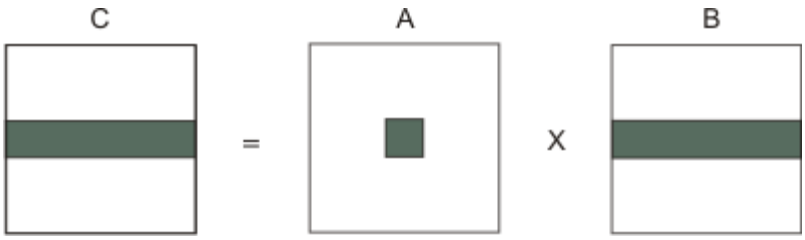
Previous Next

Spatial Locality

- Instruction count can be decremented but is not the only issue
- Performance is related to memory access
- Use cache to improve performance
- Matrices are stored row major
- Fetch A[i,k] shows good spatial locality
- Fetch B[k,j] is slow
- Re-order loops

Improve Spatial Locality

```
for i := 1 to n do
for k := 1 to n do
for j := 1 to n do
c[i,j] = c[i,j] + a[i,k] * b[k,j]
endfor
endfor
endfor
```

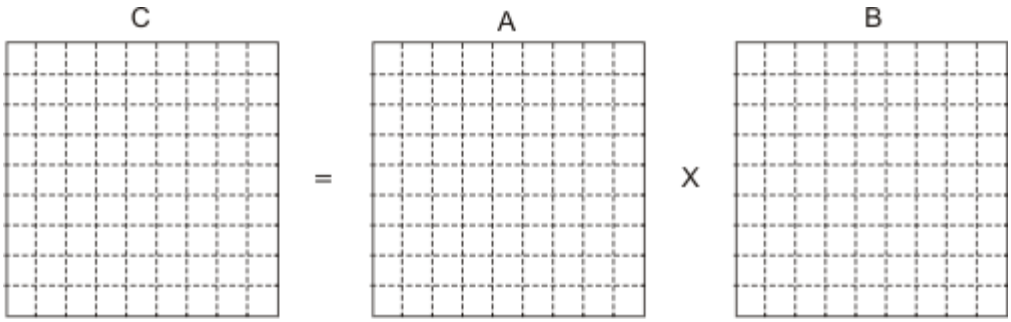


Temporal Locality

- Previous program has no temporal locality for large matrices
- Entire matrix B is fetched for each i
- If row of B or C are large it does not benefit from temporal locality
- Use sub-matrix multiplication

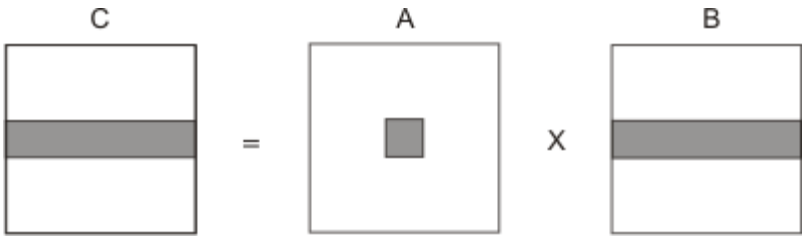
Improve Temporal Locality

```
for it := 1 to n by s do
for kt := 1 to n by s do
for jt := 1 to n by s do
for i = it to min(it+s-1, n) do
for k = kt to min(kt+s-1, n) do
for j = jt to min(jt+s-1, n) do
c[i,j] = c[i,j] + a[i,k] * b[k,j]
endfor
endfor
endfor
endfor
endfor
endfor
```



Matrix Multiplication on Vector Machine

```
for i := 1 to n do
for k := 1 to n do
c[i,1:n] = c[i,1:n] + a[i,k] * b[k,1:n]
endfor
endfor
```




```
for i = 1 to n do
for k = 1 to n do
                        ;r1 holds n and r13 holds row sizes of B and C
                        ;r2 holds address of A[i,k]
                        ;r3 holds address of B[k,1]
setvl r1                ;r4 holds address of c[i,1]
loadf f2, (r2)          ;set vactor length to n
loadv v3, (r3)          ;load A[i,k]
mpyvs v3, v3, f2        ;load B[k,1:n]
loadv v4, (r4)          ;A[i,k]*B[k,1:n]
addvv v4, v4, v3        ;load C[i,1:n]
strorev v4, (r4)        ;update C[i,1:n]
addi r2, r2, #4         ;store C[i,1:n]
add r3, r3, r13         ;point to A[i,k+1]
endfor                  ;point to B[k+1,1]
add r4, r4, r13
                        ;point to C[i+1,1]
endfor
```

Previous Next

Load and store can be floated out of k loop

```
for i = 1 to n do
setvl r1                ;set vactor length to n
loadv v4, (r4)          ;load C[i,1:n]
for k = 1 to n do
loadf f2, (r2)          ;load A[i,k]
loadv v3, (r3)          ;load B[k,1:n]
mpyvs v3, v3, f2        ;A[i,k]*B[k,1:n]
addvv v4, v4, v3        ;update C[i,1:n]
addi r2, r2, #4         ;point to A[i,k+1]
add r3, r3, r13         ;point to B[k+1,1]
endfor
strorev v4, (r4)        ;store C[i,1:n]
add r4, r4, r13         ;point to C[i+1,1]
endfor
```

Strip-mining

If n is larger than vector size, the code will not work.
To handle the general case the vector must be divided into strips of size m where m is no longer than a vector register. Assuming m=64

```
for i := 1 to n do
for k := 1 to n do
for js := 0 to n-1 by 64 do
vl := min(n-js, 64)
c[i,js+1:js+vl] = c[i,js+1:js+vl] +
a[i,k] * b[k,js+1:js+vl]
endfor
endfor
endfor
```

Shared Memory Model

- Discover iterations which can be executed in parallel
- Master processor executes task upto the parallel loop
    - Fork tasks for each of processor
    - Synchronize at the end of the loop

One way to parallelize matrix multiplication is:

```
for i := 1 to n do
for k := 1 to n do
doall j := 1 to n do
c[i,j] = c[i,j] + a[i,k] * b[k,j]
endall
endfor
endfor
```