Automatic Parallelization

- Start with sequential programming model
- Let the compiler attempt to find parallelism
  - It can be done...
  - We will look at one of the success stories
- Commonly used for SIMD computing – vectorization
  - Useful for MIMD systems, also – concurrentization
- Often done with FORTRAN
  - But, some success can be achieved with C
    (Compiler address disambiguation is more difficult with C)

Data Dependence

- Independence \Rightarrow Parallelism
  OR, dependence inhibits parallelism
  - S1: A=B+C
  - S2: D=A*2
  - S3: A=E+F
- True Dependence (RAW):
  - S1 \delta S2
- Antidependence (WAR):
  - S2 \delta S3
- Output Dependence (WAW):
  - S1 \delta S3

Outline

- Automatic Parallelization
- Vector Architectures
  - Cray-1 case study
- Data Parallel Programming
  - CM-2 case study
- CUDA Overview (separate slides)
- Readings
Data Dependence Applied to Loops

- S1 & S2
  - do I=1,2
  - S1: A(I)=B(I)+C(I)
  - S2: D(I)=A(I-1)
  - Dependence involving A, but read occurs on next loop iteration
  - Loop carried dependence
- S2 & S1
  - Antidependence involving A, write occurs on next loop iteration
  - do I=1,2
  - S1: A(I)=B(I)+C(I)
  - S2: D(I)=A(I+1)
  - if there exist i_1, i_2 where 1 \leq i_1 < i_2 \leq N and f(i_1) = g(i_2)

Loop Carried Dependence

- Definition
  - do I=1, N
  - S1: X(f(i)) = F(...)
  - S2: A = X(g(i)) ...

- Loop carried dependence if there exist i_1, i_2 where 1 \leq i_1 < i_2 \leq N and f(i_1) = g(i_2)
- However, if (for example)
  - f(i) = c*I + j and g(i) = d*I + k
  - there are methods for detecting dependence.

Loop Carried Dependences

- GCD test
  - do I=1, N
  - S1: X(c*I + j) = F(...)
  - S2: A = X(d*I + k) ...

- f(i) = g(i) if c*I + j = d*I + k
- This has a solution iff \text{gcd}(c,d) | k-j

Vector Code Generation

- In a vector architecture, a vector instruction performs identical operations on vectors of data
- Generally, the vector operations are independent
  - A common exception is reductions
- In general, to vectorize:
  - There should be no cycles in the dependence graph
  - Dependence flows should be downward \Rightarrow some rearranging of code may be needed.

Vector Code Generation: Example

- do I=1, N
  - S1: A(I) = B(I)
  - S2: C(I) = A(I) + B(I)
  - S3: E(I) = C(I+1)

- Construct dependence graph
  - S1: \delta
  - S2: \delta
  - S3: \delta

- Vectorizes (after re-ordering S2: and S3: due to antidependence)
  - S1: A(I) = B(I)
  - S2: E(I+1) = C(I+1)
  - S3: C(I) = A(I) + B(I)

Multiple Processors (Concurrentization)

- Often used on outer loops
- Example
  - do I=1, N
  - do J=1, N
  - S1: A(I,J) = B(I,J) + C(I,J)
  - S2: C(I,J) = D(I,J)/2
  - S3: E(I,J) = A(I,J-1)**2 + E(I,J-1)

- Data Dependences & Directions
  - S1 \& S3
  - S1 \& S2
  - S3 \& S3
- Observations
  - All dependence directions for I loop are = \Rightarrow iterations of the I loop can be scheduled in parallel
Scheduling

- Data Parallel Programming Model
  - SPMD (single program, multiple data)
- Compiler can pre-schedule:
  - Processor 1 executes 1st N/P iterations,
  - Processor 2 executes next N/P iterations
  - Processor P executes last N/P iterations
- Pre-scheduling is effective if execution time is nearly identical for each iteration
- Self-scheduling is often used:
  - If each iteration is large
  - Time varies from iteration to iteration
    - iterations are placed in a "work queue"
      - a processor that is idle, or becomes idle takes the next block of work from the queue (critical section)

Code Generation with Dependences

do I = 2, N
S1: A[I] = B[I] + C[I]
S2: C[I] = D[I] * 2
end do

- Data Dependences & Directions
  - S1  S2
  - S1  S3
  - S2  S3
- Parallel Code on N-1 Processors
    - signal(I)
  - S2: C[I] = D[I] * 2
    - if (I > 2) wait(I-1)
- Observation
  - Weak data-dependence tests may add unnecessary synchronization.
    - Good dependence testing crucial for high performance

Reducing Synchronization

do I = 1, N
S1: A[I] = B[I] + C[I]
S3: SUM = SUM + A[I]
end do

- Parallel Code: Version 1
  - do I = p, N, P
    - if (I > 1) wait(I-1)
    - S3: SUM = SUM + A[I]
  - signal(I)

Vectorization vs Concurrentization

- When a system is a vector MP, when should vector/concurrent code be generated?
  - do J = 1,N
    - do I = 1,N
      - S1: A(I,J+1) = B(I,J) + C(I,J)
      - S2: D(I,J) = A(I,J) * 2
    - end do
  - end do
  - Parallel & Vector Code: Version 1
    - doacross J = 1,N
      - S1: A(1:N,J+1) = B(1:N,J)+C(1:N,J)
        - signal(J)
      - if (J > 1) wait(J-1)
      - S2: D(1:N,J) = A(1:N,J) * 2
    - end do

Reducing Synchronization, contd.

- Parallel Code: Version 2
  - SUMX(p) = 0
  - do I = p, N, P
    - signal(I)
    - if (J > 1) wait(J-1)
    - S2: D(I,J) = A(I,J) * 2
    - S3: SUMX(p) = SUMX(p) + A[I]
  - barrier synchronize
    - add partial sums

- Parallel & Vector Code: Version 2
  - dforall I = 1,N
    - S1: A(1:N,J+1) = B(1:N,J)+C(1:N,J)
    - signal(J)
    - if (J > 1) wait(J-1)
    - S2: D(1:N,J) = A(1:N,J) * 2
  - end do
Summary

- Vectorizing compilers have been a success
- Dependence analysis is critical to any auto-parallelizing scheme
  - Software (static) disambiguation
  - C pointers are especially difficult
- Can also be used for improving performance of sequential programs
  - Loop interchange
  - Fusion
  - Etc. (see add’l slides at end of lecture)

Cray-1 Architecture

- Circa 1976
- 80 MHz clock
  - When high performance mainframes were 20 MHz
- Scalar instruction set
  - 16/32 bit instruction sizes
  - Otherwise conventional RISC
- 8 S register (64-bits)
- 8 A registers (24-bits)
- In-order pipeline
  - Issue in order
  - Can complete out of order (no precise traps)

Cray-1 Vector ISA

- 8 vector registers
  - 64 elements
  - 64 bits per element (word length)
  - Vector length (VL) register
- RISC format
  - Vi ← Vj OP Vk
  - Vi ← mem(Aj, disp)
- Conditionals via vector mask (VM) register
  - VM ← Vi pred Vj
  - Vi ← V2 conditional on VM

Vector Example

Do 10 i=1,looplength
   a(i) = b(i) * x + c(i)
10  continue

A1 ← looplength  ; initial values
A2 ← address(a)  ; for the array
A3 ← address(b)  ;
A4 ← address(c)  ;
A5 ← 0            ; index value
A6 ← 64          ; max hardware VL
S1 ← x            ; scalar x in register S1
VL ← A1          ; set VL – performs mod function
BrC     done, A1<=0    ; branch if nothing to do
more:        V3 ← A4,A5        ; load c indexed by A5 – addr mode not in Cray-1
V1 ← A3,A5        ; load b indexed by A5
V2 ← V1 * S1      ; vector times scalar
V4 ← V2 + V3      ; add in c
A2,A5 ← V4           ; store to a indexed by A5
A7 ← VL          ; read actual VL
A1 ← A1 – A7     ; remaining iteration count
VL ← A6 . set VL for next iteration
BrC     more, A1>0  ; branch if more work

Compare with Scalar

Do 10 i=1,looplength
   a(i) = b(i) * x + c(i)
10  continue

2 loads
1 store
1 branch
1 index increment (at least)
1 loop count increment
4-wide superscalar ⇒ up to 1 FP op per cycle
vector, with chaining ⇒ up to 2 FP ops per cycle (assuming mem b/w)
Also, in a CMOS microprocessor would save a lot of energy

Vector Conditional Loop

do 80 i = 1,looplen
   if (a(i).eq.b(i)) then
      c(i) = a(i) + e(i)
   endif
80 continue

V1 ← A1            ; load a indexed by A5 – addr mode not in Cray-1
V2 ← A2            ; load b(i)
VM ← V1 == V2      ; compare a and b, result to VM
V3 ← A3; VM        ; load e(i) under mask
V4 ← V1 + V3; VM   ; add under mask
A4 ← V4; VM       ; store to c(i) under mask
**Vector Conditional Loop**

Gather/Scatter Method (used in later Cray machines)

```plaintext
do 80 i = 1, looplen
  if (a(i).eq.b(i)) then
    c(i) = a(i) + e(i)
  endif
80    continue
```

**Thinking Machines CM1/CM2**

- Fine-grain parallelism
- Looks like intelligent RAM to host (front-end)
- Front-end dispatches "macro" instructions to sequencer
- Macro instructions decoded by sequencer and broadcast to bit-serial parallel processors

**CM Basics, contd.**

- All instructions are executed by all processors
- Subject to context flag
- Context flags
  - Processor is selected if context flag = 1
  - saving and restoring of context is unconditional
  - AND, OR, NOT operations can be done on context flag
- Operations
  - Can do logical, integer, floating point as a series of bit serial operations

**Data Parallel Programming Model**

- "Parallel operations across large sets of data"
- SIMD is an example, but can also be driven by multiple (identical) threads
  - Thinking Machines CM-2 used SIMD
  - Thinking Machines CM-5 used multiple threads

**Connection Machine Architecture**

- Nexus: 4x4, 32-bits wide
  - Cross-bar interconnect for host communications
- 16K processors per sequencer
- Memory
  - 4K mem per processor (CM-1)
  - 64K mem per processor (CM-2)
- CM-1 Processor
- 16 processors on processor chip
**Instruction Processing**

- HLLs: C* and FORTRAN 8X
- Paris virtual machine instruction set
- Virtual processors
  - Allows some hardware independence
  - Time-share real processors
  - V virtual processors per real processor
  - => 1/N as much memory per virtual processor
- Nexus contains sequencer
  - AMD 2900 bit-sliced micro-sequencer
  - 16K of 96-bit horizontal microcode
- Inst. processing:
  - 32 bit virtual machine insts (host)
  - -> 96-bit microcode (nexus sequencer)
  - -> nanocode (to processors)

**CM-2**

- re-designed sequencer; 4x microcode memory
- New processor chip
- FP accelerator (1 per 32 processors)
- 16x memory capacity (4K-> 64K)
- SEC/DED on RAM
- I/O subsystem
- Data vault
- Graphics system
- Improved router

**Performance**

- Computation
  - 4000 MIPS 32-bit integer
  - 20 GFLOPS 32-bit FP
  - 4K x 4K matrix mult: 5 GFLOPS
- Communication
  - 2-d grid: 3 microseconds per bit
  - 96 microseconds per 32 bits
  - 20 billion bits/sec
  - general router: 600 microseconds/32 bits
  - 3 billion bits/sec
- Compare with CRAY Y-MP (8 procs.)
  - 2.4 GFLOPS
  - But could come much closer to peak than CM-2
  - 246 Billion bits/sec to/from shared memory

**Outline**

- Automatic Parallelization
- Vector Architectures
  - Cray-1 case study
- Data Parallel Programming
  - CM-1/2 case study
- CUDA Overview (separate slides)
- Readings