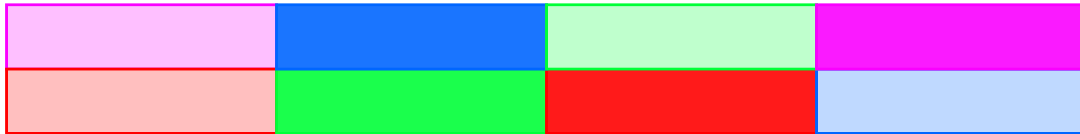
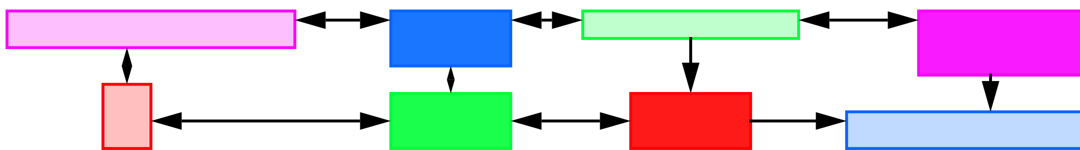
Exploiting Distribution Coherence (2)

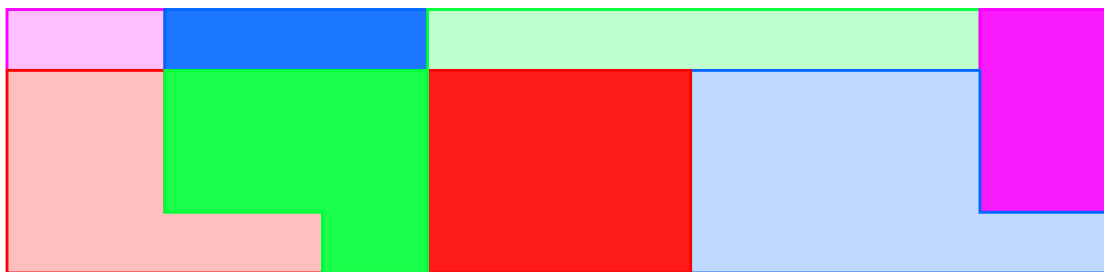
Static Work Assignment



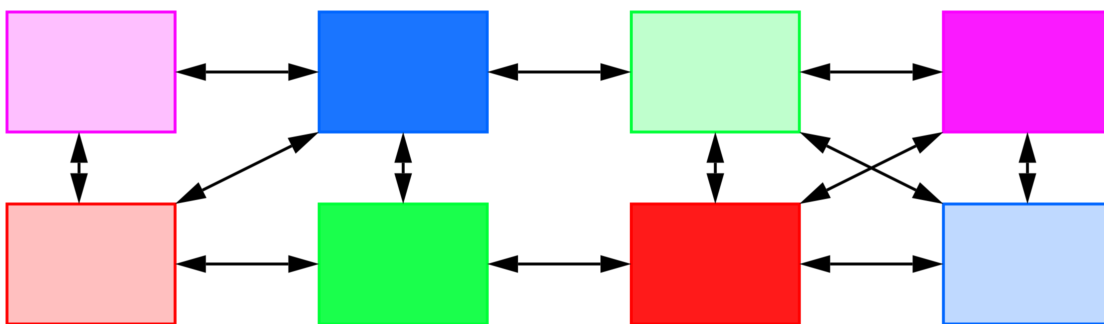
Actual Work Performed



Runtime Work Assignment



Actual Work Performed

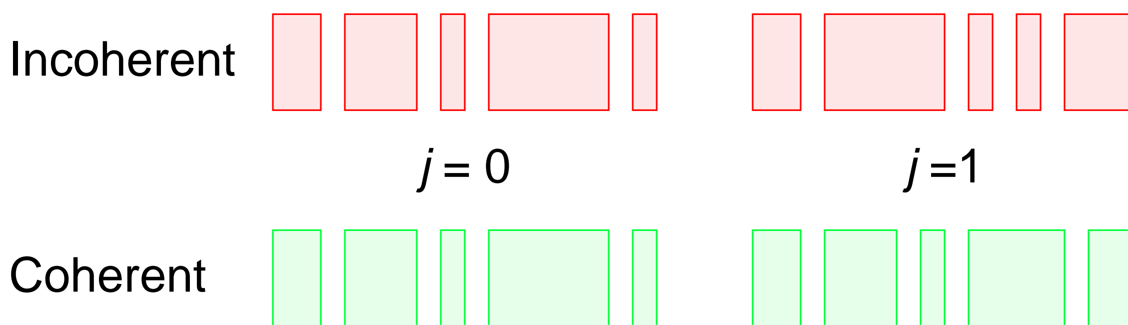


Exploiting Distribution Coherence

- ❍ Sample loop iteration execution times
- ❍ Build *work map* for each loop
- ❍ Use work map to refine allocation of work to processors
- ❍ Reduce information collection overhead by using adaptive sampling

Distribution Coherence

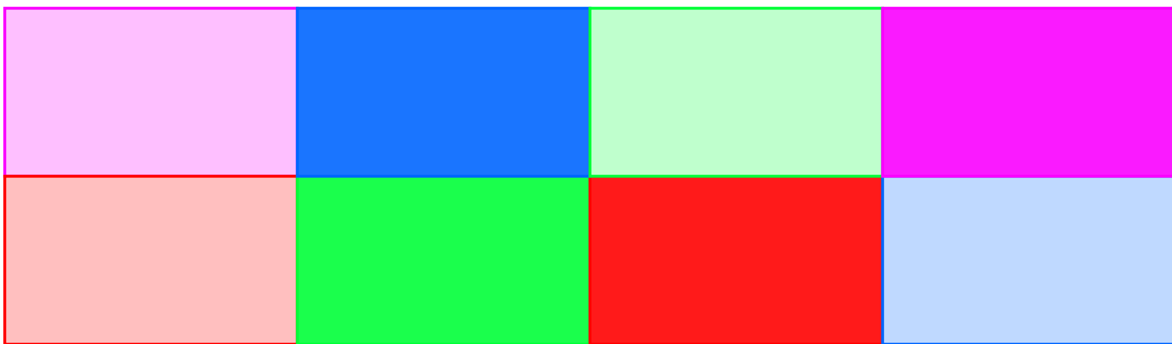
Loop A:DOALL *j* from 0 to 100
    Loop B: DOALL *k* from 0 to 100



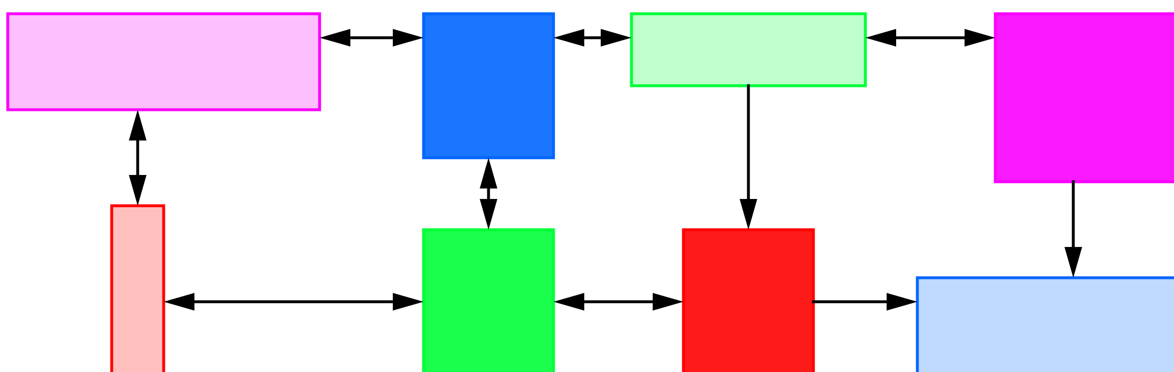
- Required for loop instance $B_{j=0}$ to predict $B_{j=1}$
  - Range coherency
  - Work distribution coherency

Static Scheduling

Static Work Assignment



Actual Work Performed



Sources of Irregular Loops

- ❍ Conditional statements
- ❍ Computed loop bounds
- ❍ While loops
- ❍ Recursion

Summary

- Dataflow languages are not the answer
  - trade parallelism for memory use
- The masses may never use anything beyond superscalar parallelism

Summary

- Optimizing compilers work
  - very well for uniprocessors (incl. superscalar)
  - quite well for the type of code found in libraries
  - not too well on vector and SIMD machines unless user codes parallel inner loops or array language
  - not well at all on distributed memory MIMD machines
- Program analysis techniques work poorly across large amounts of code
  - ```
    read i
    a(i) = 0
    ```
  - interprocedural analysis is expensive to run and to implement
  - large programs are divided up for modularity
    ```

CM-5 Fortran Compiler

- Began as port of CM-2 compiler
- Does not exploit MIMD capabilities
- Overhead of synchronization with front end makes interleaved scalar operation slow
- Very complex architecture: each CM-5 node is architecturally a CM-2

CM-2 Compiler: Limitations

- ❍ Requires inner loop parallelism (like Cray)
  - – Structure of problem may dictate otherwise
- ❍ Abstraction is not supported
  - – scope of where excludes called procedures
  - – general functions can not be called from array language statements
- ❍ SIMD architecture excludes:
  - – coarse grain parallelism
  - – parallelism across conditionals
    - ∗ programs with irregular load balance

CM-2 Fortran: Convolution Compiler

❍ Using array language eliminates need for:

– discovering parallelism

– deciding what to optimize

❍ Within array statements, many optimizations are performed very well:

– strip mining

– unrolling

– pipelining

– register reuse

– multiply-add chaining

The CM-2 CM-Fortran Compiler

❍ User must specify layout of arrays

```
     real a(100,100), b(100)
cmf$layout a(:news,:serial), b(:news)
```

❍ Conformant arrays are stored on same PE

❍ User must explicitly specify parallelism with Fortran-90 array language statements

```
a = a*c + spread(b, dim=1, ncopies=100)
do i = 1, 100
  b(i) = b(i) * x
end do
```

❍ Sequential code executed on front end

❍ Optimization only considers conformant array statements within the same basic block

Vectorization Inhibitors (cont.)

- backward branches (other than to loop entry)
- an external branch into the loop
- references to partial-word data types, including **`character`**
- references to non-vector intrinsic functions
- cross-iteration dependencies

```
a(i) = a(i-1)
```

- “ambiguous” subscript references

```
a(i) = a(i) * b(p(i))
a(i) = a(i) * a(2*i + j*7)
```

A loop is not vectorized if it contains:

- another loop
- a procedure call
- any I/O
- a **`return`**, **`stop`**, or **`pause`** statement
- computed **`goto`**, assigned **`goto`**, or 3-way **`if`**

Cray CF77 Vectorizing Transformations

- inlining

```
do i = 1, n ....................................serial:modify(a)
  call foobar(a,i)
end do

do i = 1, n ....................................parallel
  a(i) = a(i) + 1
end do
```

- nested conditionals

```
if (a(i) .ne. 0) then
  if (mod(i,7) .eq. 0) then
    a(i) = a(i) * b(i)
  else
    a(i) = a(i) / c(i)
  end if
else
  a(i) = -1.0
end if
```

- **C$DIR IVDEP**

Case Studies: Cray Fortran (CF77)

- no interprocedural analysis
- strip mining of vectors

```
a(i) = b(i)* c(i)
```

- idiom recognition: reduction

```
sum = sum + a(i)
```

- idiom recognition: search

```
if (a(i) = 0) then exit
```

- scalar expansion

```
do i = 1, n ......................................serial:output(t)
  t = a(i)*b(i)
  c(i) = c(i)/t
end do

do i = 1, n ......................................parallel
  t(i) = a(i)*b(i)
  c(i) = c(i) / t(i)
end do
```

Current Research Topics

- ❍ Symbolic analysis of subscripts
- ❍ Non-array aggregates
- ❍ Irregular problems
- ❍ Interprocedural optimizations
- ❍ Goal determination
- ❍ Composability of transformations
- ❍ Transformations for parallelism

Other Loop Transformations

- ❍ loop unrolling
- ❍ loop fusion
- ❍ loop skewing
- ❍ loop reversal
- ❍ loop peeling
- ❍ loop tiling

Loop Transformations

❍ Loop interchange

```
do i = 1, n
  do j = 2, m
    a(i,j) = a(i, j-1) * b(i,j)
  end do
end do

do j = 2, m
  do i = 1, n
    a(i,j) = a(i, j-1) * b(i,j)
  end do
end do
```

– Also used to:

- move parallelism to outer loop for multiprocessors
- improve register/cache/TLB locality

Loop Transformations

❍ Loop Distribution

```
do i = 1, n
  a(i) = a(i) * b(i)
  c(i) = c(i) * c(i-1)
end do

do i = 1, n
  a(i) = a(i) * b(i)
end do
do i = 1, n
  c(i) = c(i) * c(i-1)
end do
```

– Also used to:

  * create perfect loop nests
  * improve register/cache/TLB locality due to shorter code length

Vectorization and Parallelization

❍ strip mining

```
do i = 1, n
  a(i) = b(i) * c(i)
end do

do si = 1, n, 64
  do i = si, si+63
    a(i) = b(i) * c(i)
  end do
end do
```

– also used to combine message send operations on Distributed MIMD

Dependence Vectors

Dependence Vectors and Parallelism

- Dataflow analysis can not handle aggregates

```
do i1 = q1, r1
  do i2 = q2, r2
    a(i1, i2) = a(i1, i2-1) + b(i2, i1*2)
  end do
end do
```

- Question: what iterations does iteration $\bar{x}$ depend upon?

Dataflow Analysis

❍ Within a basic block, reveals instruction level parallelism:

```
a = 7 ................... d = 9
b = a * c
a = 9
```

❍ For global dataflow analysis:

– build Control Flow Graph (CFG) of basic blocks

– summarize write behavior for non-local variables in each basic block:

∗ kill(x) means x was written in the block

∗ gen(x) means x was written but not overwritten

– use these to find IN[B] and OUT[B], the set of writes (“definitions”) that “reach” the beginning and end of basic block B.

Dependency Analysis

Motivation: understand the essential ordering relationships of a computation.

❍ flow dependence (write->read)

```
a = 7
b = a * c
```

❍ anti-dependence (read->write)

```
b = a * c
a = 7
```

❍ output dependence (write->write)

```
a = 7
d = 9
b = a * c
a = 9
```

❍ Similar notions pervade computer science:

– transaction processing: conflict analysis

– distributed computing: partial orders

Enabling Technologies (continued)

- ❍ Dynamic Frequency Analysis (loop count and recursion depth estimation)
  - – static estimation
  - – profiling

```
do i = 1, n ..................................count=10
  do j = 1, m ..............................count=4096
    a(i,j) = a(i,j) * b(i,j) + c(i,j)
  end do
end do
```

- ❍ Memory Reference Analysis

Enabling Technologies

❍ Dependency Analysis

```
do i = 2, n .....................................serial
  a(i) = a(i-1) * b(i)
end do
do j = 1, m ...................................parallel
  y(j) = y(j) * dydth(j)
end do
```

❍ Access Pattern Analysis (arrays)

```
do i = 1, n-1 ..............................modify a(1:n-1)
  a(i) = a(i) * x
end do
do i = n, m .....................................modify a(n:m)
  a(i) = a(i) ** y(i)
end do
```

❍ Value Analysis (value numbering and constant folding/propagation)

```
b = 2
p = 3
do i = 1, n, 2
  a(i) = a(i*b**p).....read a(8:8n-8:8), write a(1,n-1,2)
end do
```

Scope of Transformations

- local (straight-line code or *basic block*)

```
x(i) = x(i) * y(n-i+1) + dxdt**2.5
i = i+1
j = mod(i, SIZE)
```

- (perfect) loop nest

```
do i = 1, n
  do j = 1, m
     a(i,j) = a(i,j) * b(j,i) + c(i)
  end do
end do
```

- global (entire procedure)

```
recursive integer function fib(i)
if (i <= 1) then
  fib = 1
else
  fib = fib(i-1) + fib(i-2)
end if
return
```

- interprocedural (entire file or program)

Compiler Technology: Transformations

- Loop Transformations
  - overhead reduction
  - redundancy elimination (common subexpressions)
  - parallelism exposure/reorganization
  - locality
- Redundancy Elimination
- Procedure Call Overhead Reduction
- Memory Usage Optimization
- Static Scheduling
- Partial Evaluation
- Array Optimizations for Parallelism

Hardware Issues for Compilation

- ❍ Minimizing dynamic instruction count
  - – eliminating redundant computations (CSE)
  - – compile-time evaluation
  - – reducing loop, procedure overhead
- ❍ Maximizing use of hardware parallelism
  - – finding and scheduling inner/outer loop parallelism
  - – minimizing load imbalance (FUs, PEs, network)
- ❍ Optimizing use of memory hierarchy
  - – register, cache, TLB, vector register, PE, subnet
  - – locality, reuse, alignment, stride, loop coherence
- ❍ Optimizing network utilization

Target Machines

- ❍ Pipelined (Sparc 1)
- ❍ Superscalar (RS 6000, HP PA)
- ❍ Vector (Cray C-90 node)
- ❍ Shared Memory MIMD Vector (Cray C-90)
- ❍ SIMD (CM-2)
- ❍ Shared Memory MIMD (Sequent, Butterfly)
- ❍ Distributed Memory MIMD (Ncube 2)
- ❍ Distributed Memory MIMD with Distributed Memory SIMD Vector nodes (CM-5)

Fortran-90 Array Language (cont.)

❍ Shift Operators

```
r   = c1*cshift(x, dim=1, shift=-1)
    + c2*x
    + c3*cshift(x, dim=1, shift=+1)
```

❍ Masking

```
where (r < ratio)
  r = r*5
elsewhere
  r = -r+x
end where
```

Fortran 90 Array Language

```
real a(100,100),b(100,100),c(100,100)
logical mask(100,100)
real x, y, z
```

- Data-parallel Array Operations

```
a = 5
b = 7
c = a * b
```

- Array Section Operations

```
a(1,:) = 9
b(40:60,30:70) = 4
c(1:50,1:50)=c(1:50,1:50)*b(11:60,11:60)
```

- Reduction and Spread Operators

```
x = maxval(c)
a = spread(c(1,:), dim=1, ncopies=100)
```

Research Languages ....(after 8)

- Fidil [Semenzato and Hilfinger, UCB]
- Functional Languages
  - Id [Arvind, MIT; Culler et al, UCB]
  - Sisal [LLNL]
- Coordination Languages
  - Strand and PCN [Chandy and Taylor, Cal Tech]
  - Linda [Gelernter, Yale]
- Extended Sequential Languages
  - Fortran D [Kennedy, Rice]
  - HPF Fortran
  - C*, Split-C, C-Linda, Dataparallel C

Commercial Source Languages

- Fortran 77
- Fortran 90
  - Connection Machine Fortran
- C

Organization

- ❍ Target machines
- ❍ Source languages
- ❍ Compiler Technology: State of the Art
  - – Dependence Analysis
- ❍ Compiler Case Studies: State of the Practice
  - – Cray Fortran 77
  - – Connection Machine Fortran (CM2 and CM5)
  - – HP PA Compiler
- ❍ Analysis and Predictions

Vectorizing and Parallelizing Compiler Technology

David F. Bacon
University of California, Berkeley