Parallelizing Compilers for Multicores

Course Offered at the Universitat Politècnica de Catalunya

Rudi (Rudolf) Eigenmann
Purdue University
School of Electrical and Computer Engineering
Computing Research Institute
Summer 2010
www.ece.purdue.edu/~eigenman/app/
How to get to Purdue University
Course Schedule

Parallelizing Compilers for Multicore

During the period 7th June– 18th June

June 7th - 11th, 10:00 - 13:00 C6-E101
June 14th - 18th, 10:00 - 13:00 C6-E101

Grading will be based on several in-class exercises and class interaction

Office hours: by appointment eigenman@purdue.edu
Course Content

• Introduction and motivation
• Detecting parallelism
• Mapping parallelism to the machine
Optimizing Compilers are the Center of the Universe

Translate increasingly advanced human interfaces onto increasingly sophisticated target machines

Today

Fortran
C, Java

Workstation
Multicores
NOW

Tomorrow

Problem Specification Language

Globally Distributed/Parallel Resources

Optimizing compilers are of particular importance where performance matters most. Hence our focus on High-Performance Computing.
Issues in Optimizing / Parallelizing Compilers

The Goal:

• We would like to run standard (C, Java, Fortran) programs on parallel computers

leads to the following high-level issues:

• How to detect parallelism?
• How to map parallelism onto the machine?
• How to create a good compiler architecture?
Detecting Parallelism

- Program analysis techniques
- Data dependence analysis
- Dependence removing techniques
- Parallelization in the presence of dependences
- Runtime dependence detection
Mapping Parallelism onto the Machine

• Exploiting parallelism at many levels
  – Multiprocessors and multi-cores (our focus)
  – Distributed memory machines (clusters or global networks)
  – Heterogeneous architectures
  – Instruction-level parallelism
  – Vector machines

• Locality enhancement
Parallelizing Compiler Books and Survey Papers

Books:
- Utpal Banerjee: several books on Data Dependence Analysis and Transformations

Survey Papers:
Course Approach

There are many schools on optimizing compilers. Our approach is *performance-driven*

Initial course schedule:

- Blume study - the simple techniques
- The Cedar Fortran Experiments
- Analysis and Transformation techniques in the Cetus compiler
- Additional transformations (for GPGPUs and other architectures)
The Heart of Automatic Parallelization

Data Dependence Testing

If a loop does not have data dependences between any two iterations then it can be safely executed in parallel.

In science/engineering applications, loop parallelism is most important. In non-numerical programs other control structures are also important.
Data Dependence Tests: Motivating Examples

**Loop Parallelization**
Can the iterations of this loop be run concurrently?

```plaintext
DO i=1,100,2
   B(2*i) = ...
   ... = B(2*i) + B(3*i)
ENDDO
```

**Statement Reordering**
Can these two statements be swapped?

```plaintext
DO i=1,100,2
   B(2*i) = ...
   ... = B(3*i)
ENDDO
```

DD testing to detect parallelism

DD testing is important not just for detecting parallelism

A data dependence exists between two data references iff:
- both references access the same storage location
- at least one of them is a write access
This course would now be finished if:

- the mathematical formulation of the data dependence problem had an accurate and fast solution, and
- there were enough loops in programs without any data dependences, and
- dependence-free code could be executed by today’s multicores directly and efficiently.

There are enough hard problems to fill several courses!
Part I: Performance of Basic Automatic Program Parallelization
15 Years of Parallelizing Compilers

A Performance study at the beginning of the 90es (Blume study)

Analyzed the performance of state-of-the-art parallelizers and vectorizers using the Perfect Benchmarks.

Overall Performance

![Overall Performance Chart]
Performance of Individual Techniques

Missing Bars indicate that no measurement was obtained

1. Vector-Concurrent
2. Recurrences
3. No Synchronizations
4. No Induction Variables
5. No Scalar Expansion
6. No Forward Substitution
7. No Reductions
8. No Loop Interchanges
9. No Stripmining

Codes

Speedup

R. Eigenmann, Parallelizing Compilers for Multicores, Summer 2010
Transformations measured in the “Blume Study”

- Scalar expansion
- Reduction parallelization
- Induction variable substitution
- Loop interchange
- Forward Substitution
- Stripmining
- Loop synchronization
- Recurrence substitution
Scalar Expansion

We assume a shared-memory model:
- by default, data is shared, i.e., all processors can see and modify it
- processor share the work of parallel loops

- DO j=1,n
  t = a(j)+b(j)
  c(j) = t + t^2
  ENDDO

Privatization

- DO PARALLEL j=1,n
  PRIVATE t
  t = a(j)+b(j)
  c(j) = t + t^2
  ENDDO

Expansion

- DO PARALLEL j=1,n
  t0(j) = a(j)+b(j)
  c(j) = t0(j) + t0(j)^2
  ENDDO
Parallel Loop Syntax and Semantics

OpenMP:

```c
!$OMP PARALLEL PRIVATE(<private data>)
  <preamble code>
!$OMP DO
  DO i = ilow, iup
    <loop body code>
  ENDDO
!$OMP END DO
  <postamble code>
!$OMP END PARALLEL
```

- executed by all participating processors (threads) exactly once
- work (iterations) shared by participating processors (threads)
Reduction Parallelization

```
DO j=1,n
    sum = sum + a(j)
ENDDO

!$OMP PARALLEL, PRIVATE (s)
s = 0
!$OMP DO
    DO j=1,n
        s = s + a(j)
    ENDDO
!$OMP ENDDO
!$OMP ATOMIC
    sum = sum + s
!$OMP END PARALLEL

!$OMP PARALLEL DO
    !$OMP+REDUCTION(+:sum)
    DO j=1,n
        sum = sum + a(j)
    ENDDO
    sum = sum + s
!$OMP END PARALLEL
```
Induction Variable Substitution

ind = ind0
DO j = 1,n
  a(ind) = b(j)
  ind = ind+k
ENDDO

ind = ind0
DO PARALLEL j = 1,n
  a(ind0+k*(j-1)) = b(j)
ENDDO

Note, this is the reverse of strength reduction, an important transformation in classical (code generating) compilers.

real d(20,100)
DO j=1,n
  d(1,j)=0
ENDDO

loop:
  ...
  R0 ← &d+20*j
  (R0) ← 0
  ...
  jump loop

R0 ← &d
loop:
  ...
  (R0) ← 0
  ...
  R0 ← R0+20
  jump loop
Forward Substitution

\[ m = n + 1 \]
\[ \ldots \]
\[ \text{DO } j=1,n \]
\[ \quad a(j) = a(j+m) \]
\[ \text{ENDDO} \]

\[ a = x \times y \]
\[ b = a + 2 \]
\[ c = b + 4 \]

dependences

\[ m = n + 1 \]
\[ \ldots \]
\[ \text{DO } j=1,n \]
\[ \quad a(j) = a(j+n+1) \]
\[ \text{ENDDO} \]

\[ a = x \times y \]
\[ b = x \times y + 2 \]
\[ c = x \times y + 6 \]

no dependences
There are many variants of stripmining (sometimes called *loop blocking*)
Loop Synchronization

DO j=1,n
    a(j) = b(j)
    c(j) = a(j)+a(j-1)
ENDDO

DOACROSS j=1,n
    a(j) = b(j)
    post(current_iteration)
    wait(current_iteration-1)
    c(j) = a(j)+a(j-1)
ENDDO
Recurrence Substitution

DO j=1,n
  a(j) = c0+c1*a(j)+c2*a(j-1)+c3*a(j-2)
ENDDO

call rec_solver(a(1),n,c0,c1,c2,c3)

Basic idea of the recurrence solver:

Error: (30) 0 Δa(10) Δa(10)+Δa(20) Δa(10)+Δa(20)+Δa
Loop Interchanging

\[
\begin{align*}
\text{DO } i & = 1, n \\
\text{DO } j = 1, m \\
a(i, j) & = a(i, j) + a(i, j-1) \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]

\[
\begin{align*}
\text{DO } j & = 1, m \\
\text{DO } i = 1, n \\
a(i, j) & = a(i, j) + a(i, j-1) \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]

- stride-1 references increase cache locality
  - read: increase spatial locality
  - write: avoid false sharing
- scheduling of outer loop is important (consider original loop nest):
  - cyclic: no locality w.r.t. to i loop
  - block schedule: there may be some locality
  - dynamic scheduling: chunk scheduling desirable
- impact of cache organization?
- parallelism at outer position reduces loop fork/join overhead
Effect of Loop Interchanging

Example: speedups of the most time-consuming loops in the ARC2D benchmark on 4-core machine

loop interchange applied in the process of parallelization
Execution Scheme for Parallel Loops

   - machine instruction for parallel loop
   - HW concurrency bus supports loop scheduling

```
<table>
<thead>
<tr>
<th>a=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO</td>
</tr>
<tr>
<td>i=1,n</td>
</tr>
<tr>
<td>b(i) = 2</td>
</tr>
<tr>
<td>ENDDO</td>
</tr>
<tr>
<td>b=3</td>
</tr>
</tbody>
</table>
```

```
| store #0,<a> |
| load <n>,D6 |
| sub 1,D6 |
| load &b,A1 |
| cdoall D6 |
| store #2,A1(D7.r) |
| endcdoall |
| store #3,<b> |
```

D7 is reserved for the loop variable. Starts at 0.
Execution Scheme for Parallel Loops

2. Microtasking scheme (dates back to early IBM mainframes)

problem: loop startup must be very fast

init_helper_tasks
wakeup_helpers
sleep_helpers
wakeup_helpers
sleep_helpers

microtask startup: 1 μs
pthreads startup: up to 100 μs
Compiler Transformation for the Microtasking Scheme

```
call init_microtasking() // once at program start
...
a=0
call loop_scheduler(loopsub,i,1,n,b)
b=3
```

```
subroutine loopsub(mytask,lb,ub,b)
  DO i=lb,ub
    b(i) = 2
  ENDDO
END
```

**Master task**
- loop_scheduler:
  - partition loop iterations
  - wakeup
  - call loopsub(…)
  - barrier (all flags reset)
- return

**Helper 1:** loopsub
- lb, ub
- param
- flag

**Shared data**

**Helper task**
- loop:
  - wait for flag
  - call loopsub(id,lb,ub,param)
- reset flag
Performance of Parallelization Techniques

## Compiler Evaluation (1990)

<table>
<thead>
<tr>
<th>Study</th>
<th>Test Suite</th>
<th>Measures</th>
<th>Machines</th>
<th>Compilers</th>
</tr>
</thead>
<tbody>
<tr>
<td>[56]</td>
<td>K</td>
<td></td>
<td></td>
<td>simulated</td>
</tr>
<tr>
<td>[149]</td>
<td>A</td>
<td></td>
<td></td>
<td>Cyber 203/5</td>
</tr>
<tr>
<td>[158]</td>
<td>P</td>
<td></td>
<td></td>
<td>Parafrase</td>
</tr>
<tr>
<td>[155]</td>
<td></td>
<td></td>
<td></td>
<td>Parafrase</td>
</tr>
<tr>
<td>[144, 143]</td>
<td></td>
<td></td>
<td></td>
<td>see Table 1</td>
</tr>
<tr>
<td>[150]</td>
<td></td>
<td></td>
<td></td>
<td>Cyber 205</td>
</tr>
<tr>
<td>[145]</td>
<td></td>
<td></td>
<td></td>
<td>FTN200, KAP, VAST</td>
</tr>
<tr>
<td>[151]</td>
<td></td>
<td></td>
<td></td>
<td>see Table 1</td>
</tr>
<tr>
<td>[148]</td>
<td></td>
<td></td>
<td></td>
<td>NAS 160</td>
</tr>
<tr>
<td>[153]</td>
<td></td>
<td></td>
<td></td>
<td>KAP, VAST</td>
</tr>
<tr>
<td>[152]</td>
<td></td>
<td></td>
<td></td>
<td>Cray Y-MP</td>
</tr>
<tr>
<td>[156]</td>
<td></td>
<td></td>
<td></td>
<td>KAP, fpp</td>
</tr>
<tr>
<td>[101, 154]</td>
<td></td>
<td></td>
<td></td>
<td>Alliant FX/8</td>
</tr>
<tr>
<td>[157]</td>
<td></td>
<td></td>
<td></td>
<td>VAST</td>
</tr>
</tbody>
</table>

**Notes:**
- K = Kernels
- A = Algorithms
- P = Application programs
- V = Shows rate of successfully vectorized loops
- N = Compares performance numbers of different compilers
- T = Compares transformations of different compilers
- S = Shows speed ups due to automatic parallelization
- I = Evaluates individual compiler techniques
- F = Discusses future compiler improvements

**Table 5:** Summary of compiler effectiveness studies
Compiler Evaluation (1990)

Table 4: Performance improvements of the Perfect Benchmarks. First two lines: Improvement over manually vector-optimized programs on Cray Y-SP [152]. Third and fourth line: Improvements over serial program execution on Alliant FX8 [153]. Fifth line: manual improvements over serial program execution on Alliant FX8 [154]
Improving Compiler-
Parallelized Code (1995)
- beyond basic techniques -

<table>
<thead>
<tr>
<th>Technique</th>
<th>ADM</th>
<th>ARC2D</th>
<th>BDNA</th>
<th>DYFESM</th>
<th>FLO52</th>
<th>MDG</th>
<th>MG3D</th>
<th>OCEAN</th>
<th>QCD</th>
<th>SPEC77</th>
<th>TRACK</th>
<th>TRFD</th>
</tr>
</thead>
<tbody>
<tr>
<td>privatize arrays</td>
<td>9.6</td>
<td>1.2</td>
<td>1.4</td>
<td>2.2</td>
<td>1</td>
<td>21</td>
<td>18</td>
<td>3.8</td>
<td>8.2</td>
<td>6.8</td>
<td>6</td>
<td>13.3</td>
</tr>
<tr>
<td>parallelize complex reductions</td>
<td>a</td>
<td>3.3</td>
<td>2.1</td>
<td>1.1</td>
<td>21</td>
<td>15.2</td>
<td></td>
<td></td>
<td></td>
<td>3.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>substitute generalized induction variables</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8.3</td>
<td></td>
<td></td>
<td></td>
<td>12.7</td>
</tr>
<tr>
<td>parallelize loops with non-affine array subscripts</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>PROGRAM-subroutine/loop</td>
<td>$T_{\text{best}}$</td>
<td>$T_{\text{unbest}}$</td>
<td>$T_{\text{unbest}} / T_{\text{best}}$ for loop</td>
<td>$T_{\text{unbest}} / T_{\text{best}}$ for program</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>---------------------------------</td>
<td>---------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AR2D-filerx/15</td>
<td>7.3</td>
<td>22.0</td>
<td>3.0</td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AR2D-filey/30</td>
<td>3.4</td>
<td>12.0</td>
<td>3.5</td>
<td>1.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADM-dctis/40</td>
<td>3.8</td>
<td>92.5</td>
<td>24</td>
<td>2.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADM-dvdta/40</td>
<td>3.5</td>
<td>76.5</td>
<td>21</td>
<td>1.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADM-dvta/40</td>
<td>3.8</td>
<td>78.8</td>
<td>21</td>
<td>1.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADM-dctis/40</td>
<td>2.6</td>
<td>51.7</td>
<td>20</td>
<td>1.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADM-dzmbh/30</td>
<td>2.5</td>
<td>37.6</td>
<td>15</td>
<td>1.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADM-dzmbh/60</td>
<td>3.8</td>
<td>86.8</td>
<td>23</td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADM-run/20</td>
<td>4.4</td>
<td>72.2</td>
<td>16.5</td>
<td>1.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADM-run/30</td>
<td>4.4</td>
<td>72.0</td>
<td>16.4</td>
<td>1.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADM-run/40</td>
<td>4.2</td>
<td>72.1</td>
<td>17</td>
<td>1.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADM-run/50</td>
<td>3.4</td>
<td>50.8</td>
<td>15</td>
<td>1.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADM-run/60</td>
<td>4.4</td>
<td>72.0</td>
<td>16.4</td>
<td>1.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADM-run/100</td>
<td>3.2</td>
<td>50.0</td>
<td>15.6</td>
<td>1.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADM-wecnt/40</td>
<td>2.7</td>
<td>36.2</td>
<td>13.4</td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADM-smooth/10</td>
<td>1.4</td>
<td>18.5</td>
<td>13.2</td>
<td>1.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BDNA-actfor/240</td>
<td>19.0</td>
<td>62.0</td>
<td>3.3</td>
<td>1.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DYPFSM-mxmult/10</td>
<td>19.0</td>
<td>60.0</td>
<td>3.2</td>
<td>1.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DYPFSM-solvh/20</td>
<td>11.5</td>
<td>26.5</td>
<td>2.3</td>
<td>1.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MDG-interf/1000</td>
<td>163.0</td>
<td>3792.0</td>
<td>23.2</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MDG-poten/2000</td>
<td>13.4</td>
<td>352.0</td>
<td>26.3</td>
<td>2.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MDG-intraf/1000</td>
<td>1.9</td>
<td>11.4</td>
<td>6.0</td>
<td>1.05</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MG3D-migrat/200</td>
<td>264.0</td>
<td>5226.4</td>
<td>19.8</td>
<td>19.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCEAN-acac/30</td>
<td>3.3</td>
<td>92.5</td>
<td>28</td>
<td>1.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCEAN-ocean/60</td>
<td>0.3</td>
<td>0.05</td>
<td>3.6</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCEAN-ocean/270</td>
<td>1.3</td>
<td>16.2</td>
<td>12.5</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCEAN-ocean/310</td>
<td>2.9</td>
<td>30.7</td>
<td>10.6</td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCEAN-ocean/360</td>
<td>2.7</td>
<td>25.6</td>
<td>9.5</td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCEAN-ocean/400</td>
<td>2.3</td>
<td>20.9</td>
<td>9.1</td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCEAN-ocean/420</td>
<td>2.3</td>
<td>25.6</td>
<td>11.1</td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCEAN-ocean/440</td>
<td>2.6</td>
<td>24.5</td>
<td>9.5</td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCEAN-ocean/460</td>
<td>7.7</td>
<td>103.7</td>
<td>13.5</td>
<td>1.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCEAN-ocean/490</td>
<td>4.1</td>
<td>91.4</td>
<td>22.3</td>
<td>1.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCEAN-ocean/500</td>
<td>2.2</td>
<td>48.0</td>
<td>21.8</td>
<td>1.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCEAN-ssec/40</td>
<td>2.5</td>
<td>48.0</td>
<td>19.2</td>
<td>1.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QCDe-measur/3</td>
<td>1.8</td>
<td>2.9</td>
<td>1.6</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QCDe-qqmeas/1</td>
<td>3.7</td>
<td>108.6</td>
<td>29.4</td>
<td>6.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QCDe-rotimes/2</td>
<td>3.7</td>
<td>48.4</td>
<td>13.1</td>
<td>3.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPEC77-gloop/1000</td>
<td>58.7</td>
<td>743.7</td>
<td>12.7</td>
<td>5.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPEC77-gwater/1000</td>
<td>13.5</td>
<td>248.0</td>
<td>18.4</td>
<td>2.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRACK-extend/400</td>
<td>4.0</td>
<td>48.9</td>
<td>12.2</td>
<td>2.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRACK-fitptrack/300</td>
<td>3.0</td>
<td>15.5</td>
<td>5.2</td>
<td>1.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRACK-nilfit/300</td>
<td>3.5</td>
<td>76.7</td>
<td>22</td>
<td>3.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRFD-olda/100</td>
<td>8.0</td>
<td>(174)</td>
<td>21.8</td>
<td>9.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRFD-olda/300</td>
<td>5.4</td>
<td>(85)</td>
<td>15.7</td>
<td>5.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Effect of Array Privatization
## Effect of Advanced Parallel Reductions

<table>
<thead>
<tr>
<th>PROGRAM-subroutine/loop</th>
<th>total loop execution time in seconds</th>
<th>$T_{without}/T_{best}$ for loop</th>
<th>$T_{without}/T_{best}$ for program</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_{best}$</td>
<td>$T_{without}$</td>
<td></td>
</tr>
<tr>
<td>MDG-interf/1000</td>
<td>163</td>
<td>3792.0</td>
<td>23.2</td>
</tr>
<tr>
<td>MDG-poteng/2000</td>
<td>13.4</td>
<td>352.0</td>
<td>26.3</td>
</tr>
<tr>
<td>DYFESM-mxmult/10</td>
<td>19.0</td>
<td>60.0</td>
<td>3.2</td>
</tr>
<tr>
<td>DYFESM-formr0/20</td>
<td>7.0</td>
<td>20.0</td>
<td>2.8</td>
</tr>
<tr>
<td>BDNA-actfor/240</td>
<td>19.0</td>
<td>62.0</td>
<td>3.3</td>
</tr>
<tr>
<td>BDNA-actfor/500</td>
<td>21.0</td>
<td>253.0</td>
<td>12.0</td>
</tr>
<tr>
<td>FLO52-euler/70</td>
<td>0.5</td>
<td>5.7</td>
<td>11.4</td>
</tr>
<tr>
<td>MG3D-migrat/200</td>
<td>264.0</td>
<td>5226</td>
<td>19.8</td>
</tr>
<tr>
<td>SPEC77-gloop/1000</td>
<td>58.7</td>
<td>743.7</td>
<td>12.7</td>
</tr>
<tr>
<td>SPEC77-gwater/1000</td>
<td>13.5</td>
<td>248.0</td>
<td>18.4</td>
</tr>
</tbody>
</table>
Effect of Generalized Induction Variable Substitution

<table>
<thead>
<tr>
<th>PROGRAM-subroutine/loop</th>
<th>total loop execution time in seconds</th>
<th>$T_{without}/T_{best}$ for loop</th>
<th>$T_{without}/T_{best}$ for program</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCEAN-ftrvmt/109</td>
<td>89  1377</td>
<td>15.5</td>
<td>8.3</td>
</tr>
<tr>
<td>TRFD-olda/100</td>
<td>8.0   174.2</td>
<td>21.8</td>
<td>9.3</td>
</tr>
<tr>
<td>TRFD-olda/300</td>
<td>5.4     85.3</td>
<td>15.8</td>
<td>5.0</td>
</tr>
</tbody>
</table>
Effect of Balanced Stripmining

<table>
<thead>
<tr>
<th>PROGRAM-subroutine/loop</th>
<th>total loop execution time in seconds</th>
<th>$T_{without}/T_{best}$ for loop</th>
<th>$T_{without}/T_{best}$ for program</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLO52-e_flux/10</td>
<td>3.9</td>
<td>1.5 (2D)</td>
<td>1.03</td>
</tr>
<tr>
<td>FLO52-e_flux/20</td>
<td>1.1</td>
<td>2.7 (1D)</td>
<td>1.03</td>
</tr>
<tr>
<td>FLO52-e_flux/30</td>
<td>3.9</td>
<td>1.5 (1D)</td>
<td>1.04</td>
</tr>
<tr>
<td>FLO52-d_flux/30</td>
<td>3.2</td>
<td>1.7 (1D)</td>
<td>1.03</td>
</tr>
<tr>
<td>FLO52-d_flux/38</td>
<td>0.4</td>
<td>1.25 (2D)</td>
<td>1.00</td>
</tr>
<tr>
<td>FLO52-bc_wall/30</td>
<td>1.7</td>
<td>1.6 (MV)</td>
<td>1.02</td>
</tr>
</tbody>
</table>
Effect of Increasing Parallel Loop Granularity

- A: inner loops parallel
- B: outer loops parallel
- C: outer loops fused

<table>
<thead>
<tr>
<th>Speed</th>
<th>Alliant FX80</th>
<th>Cedar</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>1.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1.5</td>
</tr>
<tr>
<td>.5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R. Eigenmann, Parallelizing Compilers for Multicores, Summer 2010
### Effect of Locality Enhancement

<table>
<thead>
<tr>
<th>PROGRAM-subroutine/loop</th>
<th>access</th>
<th>$T_{best}$</th>
<th>$T_{without}$</th>
<th>$T_{without}/T_{best}$ for loop</th>
<th>$T_{without}/T_{best}$ for program</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLO52-psmoo/40&amp;80</td>
<td>r/w</td>
<td>9.9</td>
<td>19.8</td>
<td>2.0</td>
<td>1.17</td>
</tr>
<tr>
<td>FLO52-step/20</td>
<td>r/o</td>
<td>1.7</td>
<td>2.4</td>
<td>1.4</td>
<td>1.01</td>
</tr>
<tr>
<td>ARC2D-xpenta/11</td>
<td>r/o</td>
<td>5.8</td>
<td>6.7</td>
<td>1.15</td>
<td>1.01</td>
</tr>
<tr>
<td>MDG-interf/1000</td>
<td>r/o</td>
<td>163</td>
<td>187</td>
<td>1.15</td>
<td>1.12</td>
</tr>
<tr>
<td>TRFD-olda/100 &amp; 300</td>
<td>r/o</td>
<td>13.4</td>
<td>15.91</td>
<td>1.18</td>
<td>1.13</td>
</tr>
</tbody>
</table>
### Effect of Runtime Data-Dependence Testing

<table>
<thead>
<tr>
<th>PROGRAM-subroutine/loop</th>
<th>total loop execution time in seconds</th>
<th>$T_{\text{without}} / T_{\text{best}}$ for loop</th>
<th>$T_{\text{without}} / T_{\text{best}}$ for program</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCEAN-ftrvmt/109</td>
<td>89.3</td>
<td>1376.9</td>
<td>15.4</td>
</tr>
<tr>
<td>OCEAN-csr/20</td>
<td>10.5</td>
<td>172.0</td>
<td>16.4</td>
</tr>
<tr>
<td>OCEAN-ftrvmt/116</td>
<td>10.8</td>
<td>99.5</td>
<td>9.2</td>
</tr>
<tr>
<td>OCEAN-acac/30</td>
<td>3.3</td>
<td>92.5</td>
<td>28.0</td>
</tr>
<tr>
<td>OCEAN-acac/40</td>
<td>4.0</td>
<td>79.0</td>
<td>19.8</td>
</tr>
<tr>
<td>OCEAN-sesec/30</td>
<td>2.3</td>
<td>59.3</td>
<td>25.8</td>
</tr>
<tr>
<td>OCEAN-res/20</td>
<td>3.4</td>
<td>57.7</td>
<td>17.0</td>
</tr>
</tbody>
</table>
Part II
A Catalog of Advanced Analysis and Transformation Techniques

- 1 Data-dependence testing
- 2 Parallelism enabling transformations
- 3 Techniques for multiprocessors/multicores
- 4 Techniques for heterogeneous multicores
- 5 Techniques for other architectures (vector, distributed-memory, …)
1 Data Dependence Testing

Earlier, we have considered the simple case of a 1-dimensional array enclosed by a single loop:

```
DO i=1,n
   a(4*i) = . . .
   . . . = a(2*i+1)
ENDDO
```

the question to answer: can 4*i ever be equal to 2*i+1 within i ∈[1,n] ?

In general: given
- two subscript functions f and g and
- loop bounds lower, upper.

Does

\[ f(i_1) = g(i_2) \]

have a solution such that

\[ lower \leq i_1, i_2 \leq upper ? \]
Data Dependence Tests: Concepts

Terms for data dependences between statements of loop iterations.

- **Distance (vector)**: indicates how many iterations apart are source and sink of dependence.
- **Direction (vector)**: is basically the sign of the distance. There are different notations: (<,=,>) or (-1,0,+1) meaning dependence (from earlier to later, within the same, from later to earlier) iteration.
- **Loop-carried (or cross-iteration) dependence** and **non-loop-carried (or loop-independent) dependence**: indicates whether or not a dependence exists within one iteration or across iterations.
  - For detecting parallel loops, only cross-iteration dependences matter.
  - *equal* dependences are relevant for optimizations such as statement reordering and loop distribution.
- **Data Dependence Graph**: a graph showing statements as nodes and dependences between them as edges. For loops, usually there is only one node per statement instance.
- **Iteration Space Graphs**: the un-abstracted form of a dependence graph with one node per statement instance. The statements of one loop iteration may be represented as a single node.
DDTests: doubly-nested loops

• Multiple loop indices:

DO i=1,n
  DO j=1,m
    X(a_1*i + b_1*j + c_1) = 
    ... = X(a_2*i + b_2*j + c_2)
  ENDDO
ENDDO

dependence problem:

a_1*i_1 - a_2*i_2 + b_1*j_1 - b_2*j_2 = c_2 - c_1
1 \leq i_1, i_2 \leq n
1 \leq j_1, j_2 \leq m

Almost all DD tests expect the coefficients a_x to be integer constants. Such subscript expressions are called affine.
DDTests: even more complexity

- Multiple loop indices, multi-dimensional array:

```
DO i=1,n
  DO j=1,m
    X(a_1*i_1 + b_1*j_1 + c_1, d_1*i_1 + e_1*j_1 + f_1) = ... 
    ... = X(a_2*i_2 + b_2*j_2 + c_2, d_2*i_2 + e_2*j_2 + f_2)
  ENDDO
ENDDO
```

dependence problem:

\[ a_1 * i_1 - a_2 * i_2 + b_1 * j_1 - b_2 * j_2 = c_2 - c_1 \]
\[ d_1 * i_1 - d_2 * i_2 + e_1 * j_1 - e_2 * j_2 = f_2 - f_1 \]
\[ 1 \leq i_1, i_2 \leq n \]
\[ 1 \leq j_1, j_2 \leq m \]
Data Dependence Tests: The Simple Case

Note: variables \( i_1, i_2 \) are integers → diophantine equations.

Equation \( a * i_1 - b * i_2 = c \) has a solution if and only iff

\[
gcd(a,b) \text{ (evenly) divides } c
\]

in our example this means: \( gcd(4,2) = 2 \), which does not divide 1 and thus there is no dependence.

If there is a solution, we can test if it lies within the loop bounds. If not, then there is no dependence.
Performing the GCD Test

- The diophantine equation
  \[ a_1 \cdot i_1 + a_2 \cdot i_2 + \ldots + a_n \cdot i_n = c \]
  has a solution iff \( \text{gcd}(a_1, a_2, \ldots, a_n) \) evenly divides \( c \)

Examples:
- \( 15 \cdot i + 6 \cdot j - 9 \cdot k = 12 \) has a solution \( \text{gcd}=3 \)
- \( 2 \cdot i + 7 \cdot j = 3 \) has a solution \( \text{gcd}=1 \)
- \( 9 \cdot i + 3 \cdot j + 6 \cdot k = 5 \) has no solution \( \text{gcd}=3 \)

Euklid Algorithm: find gcd(a,b)

Repeat
  - \( a \leftarrow a \mod b \)
  - swap \( a, b \)
Until \( b = 0 \) → *The resulting \( a \) is the gcd*

for more than two numbers:
  \[ \text{gcd}(a, b, c) = (\text{gcd}(a, \text{gcd}(b, c))) \]
Other DD Tests

• The GCD test is simple but not accurate
• Other tests
  – Banerjee test: accurate state-of-the-art test
  – Omega test: “precise” test, most accurate for linear subscripts
  – Range test: handles non-linear and symbolic subscripts
  – many variants of these tests
The Banerjee(-Wolfe) Test

Basic idea:
if the total subscript range accessed by \textit{ref1} does not overlap with the range accessed by \textit{ref2}, then \textit{ref1} and \textit{ref2} are independent.

\begin{verbatim}
DO j=1,100
  a(j) = ...
  ...
  = a(j+200)
ENDDO
\end{verbatim}

\begin{verbatim}
ranges accesses:
[1:100]
[201:300]
\rightarrow independent
\end{verbatim}
Banerjee(-Wolfe) Test continued

Weakness of the test:

Consider this dependence

\[
\begin{align*}
\text{DO } j=1,100 \\
a(j) &= \ldots \\
\ldots &= a(j+5) \\
\text{ENDDO}
\end{align*}
\]

ranges accessed:
[1:100]
[6:105]
→ independent?

We did not take into consideration that only loop-carried dependences matter for parallelization.

A loop-carried dependence only exists, if the reference in some iteration, \( j_1 \), conflicts with a reference in some later iteration, \( j_2 > j_1 \)
Banerjee(-Wolfe) Test continued

• Solution idea:
  for loop-carried dependences, make use of the fact that \( j \) in \( \text{ref2} \) is greater than in \( \text{ref1} \)

```fortran
DO j=1,100
  a(j) = ...  
  ... = a(j+5)
ENDDO
```

Ranges accessed by iteration \( j_1 \) and any other iteration \( j_2 \), where \( j_1 < j_2 \):

\[ [j_1], [j_1+6:105] \]

→ Independent for “>” direction

Still considering the potential dependence from \( a(j) \) to \( a(j+5) \)

This is commonly referred to as the Banerjee test with direction vectors.
Considering direction vectors can increase the complexity of the DD test substantially. For long vectors (corresponding to deeply-nested loops), there are many possible combinations of directions.

A possible algorithm:
1. try (*,*,...*) , i.e., do not consider directions
2. (if not independent) try (<,*,*,...*), (=,*,*,...*)
3. (if still not independent) try (<,<,*,...*),(<,>,*,...*),(<,=,*,...*)
   (=,<,*,...*), (=,>,*,...*), (=,=,*,...*)
   ...
   (This forms a tree)
Non-linear and Symbolic DD Testing

Weakness of most data dependence tests: subscripts and loop bounds must be affine, i.e., linear with integer-constant coefficients

Approach of the Range Test:
capture subscript ranges symbolically
compare ranges: find their upper and lower bounds by determining monotonicity. Monotonically increasing/decreasing ranges can be compared by comparing their upper and lower bounds.
The Range Test

Basic idea:
1. Find the range of array accesses made in a given loop iteration
2. If the upper(lower) bound of this range is less (greater) than the lower(upper) bound of the range accesses in the next iteration, then there is no cross-iteration dependence.

Example: testing independence of the outer loop:

```
DO i=1,n
  DO j=1,m
    A(i*m+j) = 0
  ENDDO
ENDDO
```

range of A accessed in iteration $i_x$: $[i_x*m+1:(i_x+1)*m]$  

range of A accessed in iteration $i_{x+1}$: $[(i_{x+1})*m+1:(i_{x+2})*m]$  

$ub_x < lb_{x+1} \Rightarrow$ no cross-iteration dependence
Range Test continued

Assume $f, g$ are monotonically increasing w.r.t. all $i_x$:
- find upper bound of access range at loop $k$:
  - successively substitute $i_x$ with $U_x$, $x=\{n, n-1, \ldots, k\}$
  - lowerbound is computed analogously

If $f, g$ are monotonically decreasing w.r.t. some $i_y$, then substitute $L_y$ when computing the upper bound.

Determining monotonicity: consider $d = f(\ldots, i_k, \ldots) - f(\ldots, i_k - 1, \ldots)$
- If $d>0$ (for all values of $i_k$) then $f$ is monotonically increasing w.r.t. $k$
- If $d<0$ (for all values of $i_k$) then $f$ is monotonically decreasing w.r.t. $k$

What about symbolic coefficients?
- in many cases they cancel out
- if not, find their range (i.e., all possible values they can assume at this point in the program), and replace them by the upper or lower bound of the range.

we need powerful expression manipulation and comparison utilities
we need range analysis
Range Test: handling non-contiguous ranges

```
DO i1=1,u1
  DO i2=1,u2
    A(n*i1+m*i2)) = …
  ENDDO
ENDDO
```

The basic Range Test finds independence of the outer loop if \( n \geq u2 \) and \( m=1 \)

But not if \( n=1 \) and \( m \geq u1 \)

Idea:
- temporarily (during program analysis) interchange the loops,
- test independence,
- interchange back

Issues:
- legality of loop interchanging,
- change of parallelism as a result of loop interchanging
Some Engineering Tasks and Questions for DD Test Pass Writers

- Start with the simple case: linear (affine) subscripts, single nests with 1-dim arrays. Subscript and loop bounds are integer constants. Stride 1 loop, lower bound = 1
- Deal with multiple array dims and loop nests
- Add capabilities for non-stride-1 loops and lower bounds ≠ 1
- How to deal with symbolic subscript coefficients and bounds
- Ignore dependences in private variables and reductions
- Generate DD vectors
- Mark parallel loops
- Things to think about:
  -- how to handle loop-variant coefficients
  -- how to deal with private, reduction, induction variables
  -- how to represent DD information
  -- how to display the DD info
  -- how to deal with non-parallelizable loops (IO op, function calls, other?)
  -- how to find eligible DO loops?
  -- how to find eligible loop bounds, array subscripts?
  -- what is the result of the pass? Generate DD info or set parallel loop flags?
  -- what symbolic analysis capabilities are needed?
Data-Dependence Test, References

- **Banerjee/Wolfe test**

- **Power Test**

- **Range test**

- **Omega test**

- **I Test**
2 Parallelism Enabling Techniques
Privatization

\begin{align*}
\text{DO } i=1,n \\
t & = A(i) + B(i) \\
C(i) & = t + t**2 \\
\text{ENDDO}
\end{align*}

\begin{align*}
\text{DO } j=1,n \\
t(1:m) & = A(j,1:m) + B(j) \\
C(j,1:m) & = t(1:m) + t(1:m)**2 \\
\text{ENDDO}
\end{align*}

loop-carried anti dependence

scalar privatization

array privatization

\begin{align*}
\text{!$OMP PARALLEL DO} \\
\text{!$OMP+PRIVATE(t)} \\
\text{DO } i=1,n \\
t & = A(i) + B(i) \\
C(i) & = t + t**2 \\
\text{ENDDO}
\end{align*}

\begin{align*}
\text{!$OMP PARALLEL DO} \\
\text{!$OMP+PRIVATE(t)} \\
\text{DO } j=1,n \\
t(1:m) & = A(j,1:m) + B(j) \\
C(j,1:m) & = t(1:m) + t(1:m)**2 \\
\text{ENDDO}
\end{align*}
Array Privatization

```
k = 5
DO j=1,n
   t(1:10) = A(j,1:10)+B(j)
   C(j,iv) = t(k)
   t(11:m) = A(j,11:m)+B(j)
   C(j,1:m) = t(1:m)
ENDDO
```

```
DO j=1,n
   IF (cond(j))
      t(1:m) = A(j,1:m)+B(j)
      C(j,1:m) = t(1:m) + t(1:m)**2
   ENDIF
   D(j,1) = t(1)
ENDDO
```

Capabilities needed for Array Privatization
- array Def-Use Analysis
- combining and intersecting subscript ranges
- representing subscript ranges
- representing conditionals under which sections are defined/used
- if ranges too complex to represent: overestimate Uses, underestimateDefs
Array Privatization continued

Array privatization algorithm:
• For each loop nest:
  – iterate from innermost to outermost loop:
    • for each statement in the loop
      – find definitions; add them to the existing definitions in this loop.
      – find array uses; if they are covered by a definition, mark this array section as privatizable for this loop, otherwise mark it as upward-exposed in this loop;
    • aggregate defined and upward-exposed, used ranges (expand from range per-iteration to entire iteration space); record them as Defs and Uses for this loop
Some Engineering Tasks and Questions for Privatization Pass Writers

- Start with scalar privatization
- Next step: array privatization with simple ranges (contiguous; no range merge) and singly-nested loops
- Deal with multiply-nested loops (→ range aggregation)
- Add capabilities for merging ranges
- Implement advanced range representation (symbolic bounds, non-contiguous ranges)
- Deal with conditional definitions and uses (too advanced for this course)
- Things to think about
  - what symbolic analysis capabilities are needed?
  - how to represent advanced ranges?
  - how to deal with loop-variant subscript terms?
  - how to represent private variables?
Array Privatization, References


• Zhiyuan Li, Array Privatization for Parallel Execution of Loops, Proceedings of the 1992 ACM International Conference on Supercomputing
Induction Variable Substitution

\[
\begin{align*}
\text{ind} &= k \\
\text{DO } i=1,n \\
\text{ind} &= \text{ind} + 2 \\
A(\text{ind}) &= B(i) \\
\text{ENDDO}
\end{align*}
\]

\[
\text{Parallel DO } i=1,n \\
A(k+2*i) &= B(i) \\
\text{ENDDO}
\]

This is the simple case of an induction variable.
Generalized Induction Variables

\begin{align*}
\text{ind} &= k \\
\text{DO } j=1,n &\quad \text{Parallel DO } j=1,n \\
&\quad \text{ind} = \text{ind} + j \\
&\quad A(\text{ind}) = B(j) \\
&\quad \text{ENDDO}
\end{align*}

\begin{align*}
\text{DO } i=1,n &\quad \text{DO } j=1,i \\
&\quad \text{ind}1 = \text{ind}1 + 1 \\
&\quad \text{ind}2 = \text{ind}2 + \text{ind}1 \\
&\quad A(\text{ind}2) = B(i) \\
&\quad \text{ENDDO}
\end{align*}

\begin{align*}
\text{DO } i=1,n &\quad \text{DO } j=1,i \\
&\quad \text{ind} = \text{ind} + 1 \\
&\quad A(\text{ind}) = B(i) \\
&\quad \text{ENDDO}
\end{align*}

\begin{align*}
\text{ENDDO}
\end{align*}
Recognizing GIVs

- **Pattern Matching:**
  - find induction statements in a loop nest of the form \( iv = iv + \text{expr} \) or \( iv = iv \cdot \text{expr} \), where \( iv \) is a scalar integer.
  - \( \text{expr} \) must be loop-invariant or another induction variable (there must not be cyclic relationships among IVs)
  - \( iv \) must not be assigned in a non-induction statement

- **Abstract interpretation:** find symbolic increments of \( iv \) per loop iteration

- **SSA-based recognition**
Computing Closed Form, Substituting additive GIVs

Loop structure \( L_0: \) 

```
For j: 1..ub
    ...
S_1: iv=iv+exp   L
    ...
S_2: loop using iv   L
    ...
S_3: stmt using iv   U
    ...
Rof
```

Main:

```
totalinc = FindIncrement(\( L_0 \))
Replace(\( L_0, iv \))
InsertStatement("iv = iv+totalinc")
```

For coupled GIVs: begin with independent iv.

---

**Step 1: find the increment rel. to start of loop \( L \)**

```
FindIncrement(\( L \))
inc=0
foreach s_i of type I,L
    if type(s_i)=I   inc += exp
    else /* L */   inc+= FindIncrement(s_i)
inc_after[s_i]=inc
inc_into_loop[\( L \)]= \( \sum_{i=1}^{i-1} (inc) \) ; inc may depend
return \( \sum_{i=1}^{ub} (inc) \) ; on j
```

**Step 2: substitute IV**

```
Replace(\( L, initialval \))
val = initialval
foreach s_i of type I,L,U
    if type(s_i)=L   Replace(s_i,val)
    if type(s_i)=L,I   val=initialval
        +inc_into_loop[\( L \)]
        +inc_after[s_i]
    if type(s_i)=U   Substitute(s_i.expr,iv,val)
```

Insert this statement if iv is live-out
Induction Variables, References

• B. Pottenger and R. Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. ACM Int. Conf. on Supercomputing (ICS'95), June 1995. (Extended version: Parallelization in the presence of generalized induction and reduction variables. www.ece.ecn.purdue.edu/~eigenman/reports/1396.pdf)

• Mohammad R. Haghighat, Constantine D. Polychronopoulos, Symbolic analysis for parallelizing compilers, ACM Transactions on Programming Languages and Systems (TOPLAS), v.18 n.4, p.477-518, July 1996

Reduction Parallelization

Scalar Reductions

DO i=1,n
  sum = sum + A(i)
ENDDO

Note, OpenMP has a reduction clause, only reduction recognition is needed:

!$OMP PARALLEL PRIVATE(s)
s=0
!$OMP DO
  DO i=1,n
    s=s+A(i)
  ENDDO
!$OMP ATOMIC
  sum = sum+s
!$OMP END PARALLEL

DO i=1,num_proc
  s(i)=0
ENDDO

!$OMP PARALLEL DO
!$OMP+REDUCTION(+:sum)
  DO i=1,n
    s(my_proc)=s(my_proc)+A(i)
  ENDDO
!$OMP ATOMIC
  sum=sum+s(i)
!$OMP END PARALLEL

R. Eigenmann, Parallelizing Compilers for Multicores, Summer 2010
Reduction Parallelization continued

Reduction recognition and parallelization passes:

- induction variable recognition
- reduction recognition
- privatization
- data dependence test
- reduction parallelization

compiler passes

recognizes and annotates reduction variables

for parallel loops with reduction variables, performance the reduction transformation
Reduction Parallelization

Array Reductions (a.k.a. irregular or histogram reductions)

DIMENSION sum(m)
DO i=1,n
  sum(expr) = sum(expr) + A(i)
ENDDO

Note, OpenMP 1.0 does not support such array reductions

R. Eigenmann, Parallelizing Compilers for Multicores, Summer 2010
Recognizing Reductions

• Pattern Matching:
  – find reduction statements in a loop of the form
    \( X = X \odot \text{expr} \),
    where \( X \) is either scalar or an array expression (\( a[\text{sub}] \),
    where \( \text{sub} \) must be the same on the LHS and the RHS),
    \( \odot \) is a reduction operation, such as +, *, min, max
  – \( X \) must not be used in any non-reduction statement
    in this loop (however, there may be multiple reduction
    statements for \( X \))
Performance Considerations for Reduction Parallelization

- Parallelized reductions execute substantially more code than their serial versions ⇒ overhead if the reduction ($n$) is small.
- In many cases (for large reductions) initialization and sum-up are insignificant.
- False sharing can occur, especially in expanded reductions, if multiple processors use adjacent array elements of the temporary reduction array ($s$).
- Expanded reductions exhibit more parallelism in the sum-up operation.
- Potential overhead in initialization, sum-up, and memory used for large, sparse array reductions ⇒ compression schemes can become useful.
Recurrence Substitution

\[
\text{DO j}=1,n \\
a(j) = c0+c1*a(j)+c2*a(j-1)+c3*a(j-2) \\
\text{ENDDO}
\]

call rec_solver(a,n,c0,c1,c2,c3)
Basic idea of the recurrence solver:

DO j=1,40
  a(j) = a(j) + a(j-1)
ENDDO

Issues:
- Solver makes several parallel sweeps through the iteration space (n). Overhead can only be amortized if n is large.
- Many variants of the source code are possible. Transformations may be necessary to fit the library call format → additional overhead.

Example from FLO52

DO 40 II=3,IL
I  = I  -1
DO 40 J=2,JL
  DW(I,J,N) = DW(I,J,N) -R*(DW(I,J,N) -DW(I+1,J,N))
40 CONTINUE
Loop Skewing

DO i=1,4
  DO j=1,6
    A(i,j)= A(i-1,j-1)
  ENDDO
ENDDO

 !$OMP PARALLEL DO
  DO wave=1,?
    i = ?
    j = ?
    wsize = ?
    DO k=0,wsize-1
      A(i+k,j+k)=A(i-1+k,j-1+k)
    ENDDO
  ENDDO
ENDDO

Iteration space graph:
Shared regions show wavefronts of iterations in the transformed code that can be executed in parallel.
Loop Skewing

DO i=1,4
  DO j=1,6
    A(i,j)= A(i-1,j-1)
  ENDDO
ENDDO

!$OMP PARALLEL DO
  DO wave=1,9
    i = max(5-wave,1)
    j = max(-3+wave,1)
    wsize = min(4,5-abs(wave-5))
    DO k=0,wsize-1
      A(i+k,j+k)=A(i-1+k,j-1+k)
    ENDDO
  ENDDO

Iteration space graph:
Shared regions show wavefronts of iterations in the transformed code that can be executed in parallel.
3 Techniques for Multiprocessors: Mapping parallelism to shared-memory machines
Loop Fusion

PARALLEL DO i=1,n
   A(i) = B(i)
ENDDO

PARALLEL DO i=1,n
   C(i) = A(i)+D(i)
ENDDO

PARALLEL DO i=1,n
   A(i) = B(i)
   C(i) = A(i)+D(i)
ENDDO

• Loop fusion is the reverse of loop distribution.
• reduces the loop fork/join overhead.
• Both transformations reorder computation;
  → data dependences show legality

R. Eigenmann, Parallelizing Compilers for Multicores , Summer 2010
Enforcing Data Dependence

- Criterion for correct transformation and execution of a computation involving a data dependence with vector $v : (=, \ldots <, \ldots *)$

Let $L_s$ be the outermost loop with non-“=” DD-direction:
- The direction at $L_s$ must be “<”
- $L_s$ must be executed serially

Note that a data dependence is defined with respect to an ordered (usually serial) execution. A fully parallel loop by definition does not have any cross-iteration dependence.
Loop Coalescing

PARALLEL DO i=1,n
  DO j=1,m
    A(i,j) = B(i,j)
  ENDDO
ENDDO

PARALLEL DO ij=1,n*m
  i = 1 + (ij-1) DIV m
  j = 1 + (ij-1) MOD m
  A(i,j) = B(i,j)
ENDDO

Loop coalescing
- can increase the number of iterations of a parallel loop → load balancing
- adds additional computation → overhead
Loop Interchange

Loop interchange affects:
• granularity of parallel computation (compare the number of parallel loops started)
• locality of reference (compare the cache-line reuse)
these two effects may impact the performance in the same or in opposite directions.

Loop interchange is subject to DD legality constraints.
DO j=1,m
DO i=1,n
  B(i,j) = A(i,j) + A(i,j-1)
ENDDO
ENDDO

This is basically the same transformation as striping, but followed by loop interchanging.
Loop Blocking continued

```fortran
DO j=1,m
  DO i=1,n
    B(i,j)=A(i,j)+A(i,j-1)
  ENDDO
ENDDO
```

```fortran
!$OMP PARALLEL
  DO j=1,m
    !$OMP DO
      DO i=1,n
        B(i,j)=A(i,j)+A(i,j-1)
      ENDDO
    !$OMP ENDDO NOWAIT
  ENDDO
!$OMP END PARALLEL
```

```
```

R. Eigenmann, Parallelizing Compilers for Multicores, Summer 2010
Choosing the Block Size

The block size must be small enough so that all data references between the use and the reuse fit in cache.

```
DO j=1,m
    DO k=1,block
        … (r1 data references)
        … = A(k,j) + A(k,j-d)
        … (r2 data references)
    ENDDO
ENDDO
```

Number of references made between the access A(k,j) and the access A(k,j-d) when referencing the same memory location:

\[(r1+r2+3)*d*block\]

\[\rightarrow \text{block} < \text{cachesize} / (r1+r2+2)*d\]

If the cache is shared, all processors use it simultaneously. Hence the effective cache size appears smaller:

\[\text{block} < \text{cachesize} / (r1+r2+2)*d*\text{num_proc}\]

In a program with multiply-nested loops, there can be a large number of possible program variants obtained through distribution and interchanging.
Multi-level Parallelism from Single Loops

\[
\text{DO } i=1, n \\
\text{A(i) = B(i)} \\
\text{ENDDO}
\]

\[
\text{PARALLEL DO } (\text{inter-cluster}) \ i1=1, n, \text{strip} \\
\text{PARALLEL DO } (\text{intra-cluster}) \ i=i1, \text{min}(i1+\text{strip}-1, n) \\
\text{A(i) = B(i)} \\
\text{ENDDO} \\
\text{ENDDO}
\]

Strip mining for multi-level parallelism
References

4 Techniques for Vector Machines
Vector Instructions

A vector instruction operates on a number of data elements at once.

Example: \texttt{vadd va,vb,vc,32}

vector operation of length 32 on vector registers \texttt{va}, \texttt{vb}, and \texttt{vc}

\begin{itemize}
\item \texttt{va}, \texttt{vb}, \texttt{vc} can be
\begin{itemize}
\item Special cpu registers or memory → classical supercomputers
\item Regular registers, subdivided into shorter partitions (e.g., 64bit register split 8-way) → multi-media extensions
\end{itemize}
\item The operations on the different vector elements can overlap → vector pipelining
\end{itemize}
Applications of Vector Operations

• Science/engineering applications are typically regular with large loop iteration counts. This was ideal for classical supercomputers, which had long vectors (up to 256; vector pipeline startup was costly).

• Graphics applications can exploit “multi-media” register features and instruction sets.
Basic Vector Transformation

DO i=1,n
    A(i) = B(i)+C(i) → A(1:n)=B(1:n)+C(1:n)
ENDDO

DO i=1,n
    A(i) = B(i)+C(i) → A(1:n)=B(1:n)+C(1:n)
    C(i-1) = D(i)**2 → C(0:n-1)=D(1:n)**2
ENDDO

The triplet notation is interpreted to mean “vector operation”. Notice that this is not (necessarily) the same meaning as in Fortran 90,
Distribution and Vectorization

The transformation done on the previous slide involves loop distribution. Loop distribution reorders computation and is thus subject to data dependence constraints.

The transformation is not legal if there is a lexical-backward dependence:

DO i=1,n
  A(i) = B(i)+C(i)
  C(i+1) = D(i)**2
ENDDO

Statement reordering may help resolve the problem. However, this is not possible if there is a dependence cycle.

A(1:n)=B(1:n)+C(1:n)
D(1:n)=A(1:n)+A(0:n-1)
Vectorization Needs Expansion

... as opposed to privatization

\[
\begin{align*}
\text{DO } & i=1,n \\
t & = A(i)+B(i) \\
C(i) & = t + t**2 \\
\text{ENDDO}
\end{align*}
\]

\[
\begin{align*}
\text{DO } & i=1,n \\
T(i) & = A(i)+B(i) \\
C(i) & = T(i) + T(i)**2 \\
\text{ENDDO}
\end{align*}
\]

(expansion)

\[
\begin{align*}
\text{DO } & i=1,n \\
T(1:n) & = A(1:n)+B(1:n) \\
C(1:n) & = T(1:n)+T(1:n)**2
\end{align*}
\]

(vectorization)
Conditional Vectorization

DO i=1,n
   IF (A(i) < 0) A(i)=-A(i)
ENDDO

WHERE (A(1:n) < 0) A(1:n)=-A(1:n)
Stripmining turns a single loop into a doubly-nested loop for two-level parallelism. It also needs to be done by the code-generating compiler to split an operation into chunks of the available vector length.
5 Advanced Program Analysis
Interprocedural Constant Propagation

Making constant values of variables known across subroutine calls

Subroutine A
j = 150
call B(j)
END

Subroutine B(m)
DO k=1,100
X(i)=X(i+m)
ENDDO
END

knowing that m>100 allows this loop to be parallelized
An Algorithm for Interprocedural Constant Propagation

Step 1: determine *jump functions* for all subroutine arguments

Subroutine $X(a,b,c)$
- $e = 10$
- $d = b+2$
- call `somesub(c)`
- $f = b*2$
- call this sub$(a,d,c,e,f)$
END

- $J1 = a$ (jump function of first parameter)
- $J2 = b+2$
- $J3 = \perp$ (called *bottom*, meaning non-constant)
- $J4 = 10$
- $J5 = \perp$

- Mechanism for finding jump functions: (local) forward substitution and interprocedural MAYMOD analysis.
- Here we assume jump functions are of the form $P+const$ ($P$ is a subroutine parameter of the callee).
Constant Propagation Algorithm
continued

Step 2:
• initialize all formal parameters to the value $T$ (called $top$, meaning non-yet-known)

• for all jump functions:
  – if it is $\bot$: set formal parameter value to $\bot$
  – if it is constant and the value of the formal parameter is the same constant or $T$ : set it to this constant
Constant Propagation Algorithm continued

Step 3:

1. put all formal parameters on a work queue

2. take a parameter from the queue:
   
   for all jump functions that contain this parameter:
   
   • determine the value of the target parameter of this jump function. Set it to this value, or to $\bot$ if it is different from a previously set value.
   
   • if the value of the target parameter changes, put this parameter on the queue

3. repeat 2 until the queue is empty
### Interprocedural Data-Dependence Analysis

#### Motivational examples:

- **DO i=1,n**
  ```plaintext
call clear(a,i)
ENDDO
```

- **Subroutine clear(x,j)**
  ```plaintext
  x(j) = 0
  END
  ```

- **DO i=1,n**
  ```plaintext
  a(i) = b(i)
call dupl(a,i)
ENDDO
  ```

- **Subroutine dupl(x,j)**
  ```plaintext
  x(j) = 2*x(j)
  END
  ```

- **DO i=1,n**
  ```plaintext
  a(i) = b(i)
call smooth(a,i)
ENDDO
  ```

- **Subroutine smooth(x,j)**
  ```plaintext
  x(j) = (x(j-1)+x(j)+x(j+1))/3
  END
  ```
Interproc. DD-analysis

• Overall strategy:
  – subroutine inlining
  – move loop into called subroutine
  – collect array access information in callee and use in the analysis of the caller
    → will be discussed in more detail
Interproc. DD-analysis

• Representing array access information
  – summary information
    • [low:high] or [low:high:stride]
    • sets of the above
  – exact representation
    • essentially all loop bound and subscript information is captured
  – representation of multiple subscripts
    • separate representation
    • linearized
Interproc. DD-analysis

- Reshaping arrays
  - simple conversion
    - matching subarray or 2-D → 1-D
  - exact reshaping with div and mod
  - linearizing both arrays
  - equivalencing the two shapes
    - can be used in subroutine inlining

Important: reshaping may lose the implicit assertion that array bounds are not violated!
Symbolic Analysis

• Expression manipulation techniques
  – Expression simplification/normalization
  – Expression comparison
  – Symbolic arithmetic

• Range analysis
  – Find lower/upper bounds of variable values at a given statement
    • For each statement and variable, or
    • Demand-driven, for a given statement and variable
6 Techniques Specific to Distributed-memory Machines
Execution Scheme on a Distributed-Memory Machine

Typical execution scheme:
- All nodes execute the same program
- Program uses node_id to select the subcomputation to execute on each participating processor and the data to access.

For example,

```
DO i=lb,ub
  ...
ENDDO
```

```
DO i=1,n
  ...
ENDDO
```

This is called Single-Program-Multiple-Data (SPMD) execution scheme
Data Placement

Single owner:
• Data is distributed onto the participating processors’ memories

Replication:
• Multiple versions of the data are placed on some or all nodes.
Data Distribution Schemes

numbers indicate the node of a 4-processor distributed-memory machine on which the array section is placed

1 2 3 4

block distribution

1234123412341234 ···
cyclic distribution

1 2 3 4 1 ···

block-cyclic distribution

IND(1) IND(2) IND(3) IND(4) IND(5) ···
indexed distribution

index array

Automatic data distribution is difficult because it is a global optimization.
Message Generation for single-owner placement

DO i=1,n
    B(i) = A(i)+A(i-1)
ENDDO

send (A(ub),my_proc+1)
receive (A(lb-1),my_proc-1)

DO i=lb,ub
    B(i) = A(i)+A(i-1)
ENDDO

- lb,ub determine the iterations assigned to each processor.
- array distributions assumed to match the iteration distribution
- my_proc is the current processor number

Compilers for languages such as HPF (High-Performance Fortran) have explored these ideas extensively.
Owner-computes Scheme

In general, the elements accessed by a processor are different from the elements owned by this processor as defined by the data distribution.

```
DO i=1,n
  A(i)=B(i)+B(i-m)
  C(ind(i))=D(ind2(i))
ENDDO
```

```
DO i=1,n
  send/receive what’s necessary
  IF I_own(A(i)) THEN
    A(i) = B(i)+B(i-m)
  ENDIF
  send/receive what’s necessary
  IF I_own(C(ind(i))) THEN
    C(ind(i))=D(ind2(i))
  ENDIF
ENDDO
```

- nodes execute those iterations and statements whose LHS they own
- first they receive needed RHS elements from remote nodes
- nodes need to send all elements needed by other nodes

*Example shows basic idea only. Compiler optimizations needed!*
Compiler Optimizations

for the raw owner computes scheme

- **Eliminate conditional execution**
  - combine if statements with same condition
  - reduce iteration space if possible

- **Aggregate communication**
  - combine small messages into larger ones
  - tradeoff: delaying a message enables message aggregation but increases the message latency.

- **Message Prefetch**
  - moving `send` operations earlier in order to reduce message latencies.

there is a large number of research papers describing such techniques
Message Generation for replication

- Fully parallel section w. local reads and writes
- Broadcast written data
- Fully parallel section w. local reads and writes

Optimization: reduce broadcast operations to necessary point-to-point communication

Advantages:
- Fully parallel sections with local reads and writes
- Easier message set computation (no partitioning per processor needed)

Disadvantages:
- Not data-scalable
- More write operations necessary (but, collective communication can be used)
References

Data distribution and message generation:
(there is a large number of references on these topics)


Message Generation under Replication:

7 Techniques for Instruction-Level Parallelization
Implicit vs. Explicit ILP

**Implicit ILP**: ISA is the same as for sequential programs.

- most processors today employ a certain degree of implicit ILP
- parallelism detection is entirely done by the hardware, however,
- compiler can assist ILP by arranging the code so that the detection gets easier.
Implicit vs. Explicit ILP

Explicit ILP: ISA expresses parallelism.
- parallelism is detected by the compiler
- parallelism is expressed in the form of
  - VLIW (very long instruction words): packing several instructions into one long word
  - EPIC (Explicitly Parallel Instruction Computing): bundles of (up to three) instructions are issued. Dependence bits can be specified.

Used in Intel/HP IA-64 architecture. The processor also supports predication, early (speculative) loads, prepare-to-branch, rotating registers.
Trace Scheduling
(invented for VLIW processors, still a useful terminology)

Two big issues must be solved by all approaches:
1. Identifying the instruction sequence that will be inspected for ILP.
   Main obstacle: branches
2. Reordering instructions so that machine resources are exploited efficiently.

trace selection
trace compaction
trace scheduling
Trace Selection

- It is important to have a large instruction window (block) within which the compiler can find parallelism.
- Branches are the problem. Instruction pipelines have to be flushed/squashed at branches
- Possible remedies:
  - eliminate branches
  - code motion can increase block size
  - block can contain out-branches with low probability
  - predicated execution
Branch Elimination

- Example:

```
<table>
<thead>
<tr>
<th>comp R0 R1</th>
<th>comp R0 R1</th>
</tr>
</thead>
<tbody>
<tr>
<td>bne L1:</td>
<td>beq L2:</td>
</tr>
<tr>
<td>bra L2:</td>
<td></td>
</tr>
<tr>
<td>L1: . . .</td>
<td>L1: . . .</td>
</tr>
<tr>
<td></td>
<td>. . .</td>
</tr>
<tr>
<td>L2: . . .</td>
<td>L2: . . .</td>
</tr>
</tbody>
</table>
```
Code motion can increase window sizes and eliminate subtrees.
Predicated Execution

IF (a>0) THEN
   b=a
ELSE
   b=-a
ENDIF

p = a>0 ; assignment of predicate
p: b=a ; executed if predicate true
\bar{p}: b=-a ; executed if predicate false

Predication
• increases the window size for analyzing and exploiting parallelism
• increases the number of instructions “executed”
These are opposite demands!

Compare this technique to conditional vectorization
Dependence-removing ILP Techniques

Shaded blocks of statements are independent of each other and can be executed as parallel instructions.
Speculative ILP

Speculation is performed by the architecture in various forms

- Superscalar processors: compiler only has to deal with the performance model. ISA is the same as for non-speculative processors
- Multiscalar processors: (research only) compiler defines tasks that the hardware can try execute speculatively in parallel. Other than task boundaries, the ISA is the same.

References:

Compiler Model of Explicit Speculative Parallel Execution
(Multicalar Processor)

- **Overall Execution**: *speculative threads* choose and start the execution of any predicted next thread.
- **Data Dependence and Control Flow Violations** lead to roll-backs.
- **Final Execution**: satisfies all cross-segment flow and control dependences.
- **Data Access**: Writes go to thread-private speculative storage. Reads read from ancestor thread or memory.

- **Dependence Tracking**: Data Flow and Control Flow dependences are detected directly. Lead to roll-back. Anti and Output dependences are satisfied via speculative storage.
- **Segment Commit**: Correctly executed threads (i.e., their final execution) commit their speculative storage to the memory, in sequential order.
8 OpenMP for Distributed Parallel Systems
Is OpenMP a Useful Programming Model for Distributed Processors?

- OpenMP is a parallel programming model that assumes a shared address space
  
  ```c
  #pragma OMP parallel for
  for (i=1; i<n; i++) {a[i] = b[i];}
  ```

- Why is it difficult to implement OpenMP for distributed processors?
  The compiler or runtime system will need to
  - partition and place data onto the distributed memories
  - send/receive messages to orchestrate remote data accesses
  HPF (High Performance Fortran) was a large-scale effort to do so - without success

- So, why should we try again?
  OpenMP is an easier programming (higher-productivity?) programming model. 
  OpenMP
  - allows programs to be parallelized incrementally, starting from the serial versions,
  - relieves the programmer of the task of managing the movement of logically shared data.
Baseline Translation of OpenMP to MPI Compiler

• Execution Model
  – SPMD model
    • Serial Regions are replicated on all processes
    • Iterations of parallel for loops are distributed (using static block scheduling)
  – Shared Data is allocated on all nodes
    • There is no concept of “owner” (contrast to HPF)
      There are only producers and consumers of shared data
    • At the end of a parallel loop, producers communicate shared data to potential future consumers
    • The compiler uses array section analysis for summarizing array accesses
Baseline Translation

Translation Steps:

1. Identify all shared data
2. Create annotations for accesses to shared data (use regular section descriptors to summarize array accesses)
3. Use interprocedural data flow analysis to identify potential consumers; incorporate OpenMP relaxed consistency specifications
4. Create message sets to communicate data between producers and consumers
Message Set Generation

Processor p writes array section A from l1(p) to u1(p)

Message Set at RSD vertex V1, for array A from process p to process q computed as

\[ S_{Apq} = \text{Elements of A with subscripts in the set} \]
\[ \{[l1(p),u1(p)] \cap [l2(q),u2(q)]\} \cup \{[l1(p),u1(p)] \cap [l4(q),u4(q)]\} \]
\[ \cup ([l1(p),u1(p)] \cap \{[l5(q),u5(q)]-[l3(p),u3(p)]\}) \]
Incorporating OpenMP Relaxed Memory Consistency Specifications

mydo = 1;
#pragma omp parallel
{
    /* Loop L1 */
    #pragma omp for nowait
    for(j=1;j<N;j++) {
        #pragma omp flush(myflag)
        ....myflag..
        A[j] = ...
    }
    /* Loop L2 */
    #pragma omp for nowait
    for(j=1;j<M;j++) {
        ... = A[i]...
    }
    myflag++;
}

/* Start_parallel*/

/*loop_entry for Loop L1*/
<A, lb(1,N,p), ub(1,N,p), write>
/* Loop Body for Loop L1 */
....myflag..
/* loop exit for Loop L1 */

/*loop_entry for Loop L2 */
<A, lb(1,M,p), ub(1,M,p), read>
/* Loop Body for Loop L2 */
....
/* loop exit for Loop L2 */

/*End_parallel*/
Translation of Irregular Accesses

- Irregular Access – $A[B[i]], A[f(i)]$ where $B[i]$ is a subscript array or $f(i)$ is a non-affine function of the loop index.
  - Reads: assumed the whole array accessed
  - Writes: inspect at runtime, communicate at the end of parallel loop

- A key property is Monotonicity: $i > j \rightarrow B[i] > B[j]$  
  Monotonicity is useful because it provides bounds on the irregular subscript in terms of the loop bounds
  - For $lb<i<ub$, $B[lb] < B[i] < B[ub]$  
  Monotonicity allows the compiler to
    - tighten array sections
    - avoid runtime inspection of written array sections that do not overlap
Optimizations based on Collective Communication

• Recognition of Reduction Idioms
  – Recognize program patterns that implement array reductions
    – usually: combination of parallel loop and critical section.
  – Translate them to MPI_Reduce / MPI_Allreduce functions.

• Casting sends/receives in terms of alltoall calls
  – In general, communication between producers and consumers are done using non-blocking send/recv and MPI_Wait
  – There may be insufficient distance between the production and consumption points in the program to allow overlap of computation and communication
  – When the producer-consumer relationship is many-to-many and there is insufficient distance between producers and consumers, cast the sends/recvs into a single MPI_Alltoallv call
Performance Evaluation of Baseline Translation

Platform I – Cluster of sixteen PIII 800 MHz Linux nodes, with 256 MB memory per node, connected by a commodity 100 Mbps Ethernet network.

From NAS Par. Benchmarks

From SPEC OMP2001
Performance on IBM-SP2

Platform II – Sixteen IBM SP-2 WinterHawk-II nodes connected by a high-performance switch.
Comparison with SDSM on Linux Cluster
Performance Comparison with HPF on Linux Cluster

![Graphs showing performance comparison between OpenMP and HPF on Linux Cluster for CG (CLASS B) and LU (CLASS A) on Ethernet.](image)
We can do more for Irregular Applications

L1 : #pragma omp parallel for
for(i=0;i<10;i++)
A[i] = ...

L2 : #pragma omp parallel for
for(j=0;j<20;j++)
B[j] = A[C[j]] + ...

- Subscripts of accesses to shared arrays not always analyzable at compile-time
- Baseline OpenMP to MPI translation:
  - Conservatively estimate that each process accesses the entire array
  - Try to deduce properties such as monotonicity for the irregular subscript to refine the estimate
- Still, there may be redundant communication
  - Runtime tests (inspection) are needed to resolve accesses
Inspection

- Inspection allows accesses to be differentiated (at runtime) as local and non-local accesses.
- Inspection can also map iterations to accesses. This mapping can then be used to re-order iterations:
  - first, iterations that access local data
  - then, iterations that access remote data

=> Communication of remote data can be overlapped with the computation of iterations that access local data
Loop Restructuring

- Simple iteration reordering may not be sufficient to expose the full set of possibilities for computation-communication overlap.

```c
L1 : #pragma omp parallel for
    for(i=0;i<N;i++)
        p[i] = x[i] + alpha*r[i] ;

L2 : #pragma omp parallel for
    for(j=0;j<N;j++) {
        w[j] = 0 ;
        for(k=rowstr[j];k<rowstr[j+1];k++)
            S2: w[j] = w[j] + a[k]*p[col[k]] ;
    }
```

Reordering loop L2 may still not club together accesses from different sources

```c
L1 : #pragma omp parallel for
    for(i=0;i<N;i++)
        p[i] = x[i] + alpha*r[i] ;

L2-1 : #pragma omp parallel for
    for(j=0;j<N;j++) {
        w[j] = 0 ;
    }

L2-2: #pragma omp parallel for
    for(j=0;j<N;j++) {
        for(k=rowstr[j];k<rowstr[j+1];k++)
            S2: w[j] = w[j] + a[k]*p[col[k]] ;
    }
```
Loop Restructuring continued

L1: #pragma omp parallel for
    for(i=0;i<N;i++)
        p[i] = x[i] + alpha*r[i] ;

L2-1: #pragma omp parallel for
    for(j=0;j<N;j++) {
        w[j] = 0 ;
    }

L2-2: #pragma omp parallel for
    for(j=0;j<N;j++) {
        for(k=rowstr[j];k<rowstr[j+1];k++)
            S2: w[j] = w[j] + a[k]*p[col[k]] ;
    }

L1: #pragma omp parallel for
    for(i=0;i<N;i++)
        p[i] = x[i] + alpha*r[i] ;

L2-1: #pragma omp parallel for
    for(j=0;j<N;j++) {
        w[j] = 0 ;
    }

L3: for(i=0;i<num_iter;i++)

Coalesce nested loop L2-2 to form loop L3

Reorder iterations of loop L3 to achieve computation-communication overlap

Final restructured and reordered loop

The T[i] data structure is created and filled in by the inspector
Achieving Actual Computation-Communication Overlap

• Non-blocking send/recv calls may not actually progress concurrently with computation.
  – Use a multi-threaded runtime system with separate computation and communication threads – on dual CPU machines these threads can progress concurrently.

• The compiler extracts the send/recvs along with the packing/unpacking of message buffers into a communication thread.
Communication Thread on Process $p$

- Initiate sends to process $q,r$
- Pack data and send to processes $q$ and $r$.
- Receive data from process $q$ (at $t_{recv-q}$)
- Receive data from process $r$ (at $t_{recv-r}$)

Computation Thread on Process $p$

- Execute iterations that access local data (at $t_{comp-p}$)
- Wait for receives from process $q$ to complete (at $t_{wait-q}$)
- Execute iterations that access data received from process $q$ (at $t_{comp-q}$)
- Wait for receives from process $r$ to complete (at $t_{wait-r}$)
- Execute iterations that access data received from process $r$ (at $t_{comp-r}$)

Program Timeline
Performance of Equake

Computation-communication overlap in Equake
Performance of Moldyn

Computation-communication overlap in Moldyn
Performance of CG

Computation-communication overlap in CG
Summary

OpenMP for Distributed Systems

- There is hope for easier programming models on distributed processing systems (DPS)

- OpenMP can be translated effectively onto DPS; we have used benchmarks from
  - SPEC OMP
  - NAS
  - additional irregular codes

- Caveats:
  - black-belt programmers will always be able to do better
  - advanced compiler technology is involved. There will be performance surprises
  - Larger set of and full compiler implementation are needed

=> this is ongoing work
References

