Automatic Parallelization of Programs

Sanjeev K Aggarwal
Department of Computer Science and Engineering
IIT Kanpur 208016
INDIA

REACH Symposium 2010, IIT Kanpur

October 10, 2010

Fact #1: Computer Systems

- Processors and Computer Systems are becoming more and more powerful
  - Faster and many core processors
  - High speed memory and disks
  - Large memory bandwidth
  - Low latency networks

Fact #1: Computer Systems

- Processors and Computer Systems are becoming more and more powerful
  - Faster and many core processors
  - High speed memory and disks
  - Large memory bandwidth
  - Low latency networks
- In few years time a desktop will be able to deliver a tera-flop of compute power

Fact #2: Supercomputing Applications

- Used in applications requiring very large compute time
- Important applications:
  - Engineering simulations (FM, CFD, structures)
  - Biology (Genetics, cell structure, molecular biology)
  - Nuclear physics, high energy particle physics, astrophysics, weather prediction, molecular dynamics
  - Drug design, medical industry, tomography

Fact #2: Supercomputing Applications

- Used in applications requiring very large compute time
- Important applications:
  - Engineering simulations (FM, CFD, structures)
  - Biology (Genetics, cell structure, molecular biology)
  - Nuclear physics, high energy particle physics, astrophysics, weather prediction, molecular dynamics
  - Drug design, medical industry, tomography
  - Financial services, data-mining, web services, data centers, search engines
  - Entertainment industry, animation, gaming

Systems and Applications

- These applications require enormous compute power
- The compute power is available

Systems and Applications

- These applications require enormous compute power
- The compute power is available
- Where is the CHALLENGE/GAP?

Challenge

- The biggest challenge: how do we program these machines?
- Solving these complex problems, and programming these architectures require different methodology
- Better algorithms, compilers, libraries, profilers, application tuners, debuggers etc. have to be designed
- Software systems (algorithms, compilers, libraries, debuggers) AND programmers (mainly trained in sequential programming) are unable to exploit the compute power

Challenge

- The largest user base is outside computer science domain
- Engineers and scientists should not be spending their energy in understanding machine architecture, concurrency and language issues
  - They want to solve their problems
  - They are domain experts and not system experts

High Performance Systems

Power of supercomputers comes from

- Hardware technology: faster processors and memories, more cache, low latency between devices

High Performance Systems

Power of supercomputers comes from

- Hardware technology: faster processors and memories, more cache, low latency between devices
- Multilevel architectural parallelism

Pipeline	out of order execution
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
Clusters	large number of processors on a very fast network
Multicore	lot of processors on a single chip
Grid	Cooperation of a large number of systems

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

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:
  - Write sequential code and use compilers
  - Write sequential code and use parallel APIs (MPI, OpenMP etc.)
    - Use fine grain parallelism: basic block or statement
    - Use medium grain parallelism: loop level
    - Use coarse grain parallelism: independent modules/tasks
  - Specify parallelism in parallel languages

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:
  - Write sequential code and use compilers
  - Write sequential code and use parallel APIs (MPI, OpenMP etc.)
    - Use fine grain parallelism: basic block or statement
    - Use medium grain parallelism: loop level
    - Use coarse grain parallelism: independent modules/tasks
  - Specify parallelism in parallel languages
- Expressing parallelism in programs
  - No good programming languages
  - Most applications are not multi threaded
  - Writing multi-threaded code increases software cost
  - Programmers are unable to exploit whatever little is available

State of the Art

- Current software technology is unable to handle all these issues
- Most of the programming still happens in Fortran/C/C++ using parallel APIs

State of the Art

- Current software technology is unable to handle all these issues
- Most of the programming still happens in Fortran/C/C++ using parallel APIs
- Main directions of research
  - Design of concurrent languages (X10 of IBM)
  - Construction of tools to do software development
    - Profilers, code tuning, thread checkers, deadlock/race detection
    - Parallelizing compilers
    - Better message passing libraries
  - Development of mathematical libraries

State of the Art

- Current software technology is unable to handle all these issues
- Most of the programming still happens in Fortran/C/C++ using parallel APIs
- Main directions of research
  - Design of concurrent languages (X10 of IBM)
  - Construction of tools to do software development
    - Profilers, code tuning, thread checkers, deadlock/race detection
    - Parallelizing compilers
    - Better message passing libraries
  - Development of mathematical libraries
- Dusty decks problem
  - Conversion of large body of existing sequential programs developed over last 45 years
  - Several billion lines of working code (almost debugged)
  - Manual conversion is not possible

State of the Art

- Current software technology is unable to handle all these issues
- Most of the programming still happens in Fortran/C/C++ using parallel APIs
- Main directions of research
  - Design of concurrent languages (X10 of IBM)
  - Construction of tools to do software development
    - Profilers, code tuning, thread checkers, deadlock/race detection
    - Parallelizing compilers
    - Better message passing libraries
  - Development of mathematical libraries
- Dusty decks problem
  - Conversion of large body of existing sequential programs developed over last 45 years
  - Several billion lines of working code (almost debugged)
  - Manual conversion is not possible

Therefore, RESTRUCTURING COMPILERS ARE REQUIRED

Amdahl's Law

Determines speed up

α: fraction of code in scalar
$1 - \alpha$: fraction of code which is parallelizable

1 operation per unit time in scalar unit
τ operations per unit time in parallel units

Amdahl's Law

Determines speed up

α: fraction of code in scalar
$1 - \alpha$: fraction of code which is parallelizable

1 operation per unit time in scalar unit
τ operations per unit time in parallel units

$$\text{Speed-up} = \frac{1}{\alpha + \frac{1-\alpha}{\tau}}$$

$0 \leq \alpha \leq 1$ and $\tau \geq 1$

Amdahl's Law

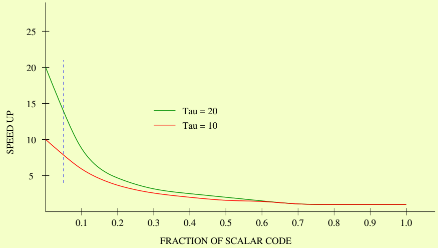
Determines speed up

α: fraction of code in scalar
$1-\alpha$: fraction of code which is parallelizable

1 operation per unit time in scalar unit
τ operations per unit time in parallel units

$$\text{Speed-up} = \frac{1}{\alpha + \frac{1-\alpha}{\tau}}$$

$0 \le \alpha \le 1$ and $\tau \ge 1$



Amdahl's Law

- Scalar component of the code is the limiting factor
- Parallel code must be greater than 90% to achieve any significant speedup
- Most of the execution time is spent in small sections of code
  - concentrate on critical sections (loops)
- Loop parallelization is the most beneficial
- Automatic parallelization must focus on the loops

Loop Optimization

- Loop unrolling, jamming, splitting
- Induction variable simplification

- Loop interchange
- Node splitting
- Loop skewing
- Conversion to parallel loops

- Inspector Executor parallelization
- Speculative parallelization
- Use APIs like OpenMP, MPI, PVM, Cuda etc.

Loop Optimization

- Loop unrolling, jamming, splitting
- Induction variable simplification

- Loop interchange
- Node splitting
- Loop skewing
- Conversion to parallel loops

- Inspector Executor parallelization
- Speculative parallelization
- Use APIs like OpenMP, MPI, PVM, Cuda etc.

- How does compiler perform these optimizations?

Parallelization: Programmers vs. Compilers

	Programmer	Compiler
Speed	Slow	Extremely fast
Working Set	Small	Very large
Accuracy	Makes mistakes	Accurate

Parallelization: Programmers vs. Compilers

	Programmer	Compiler
Speed	Slow	Extremely fast
Working Set	Small	Very large
Accuracy	Makes mistakes	Accurate
Effectiveness	Good	Poor
Preserves	Functionality	Implementation
Approach	Experiment as much as possible	Conservative

Parallelization: Programmers vs. Compilers

	Programmer	Compiler
Speed	Slow	Extremely fast
Working Set	Small	Very large
Accuracy	Makes mistakes	Accurate
Effectiveness	Good	Poor
Preserves	Functionality	Implementation
Approach	Experiment as much as possible	Conservative

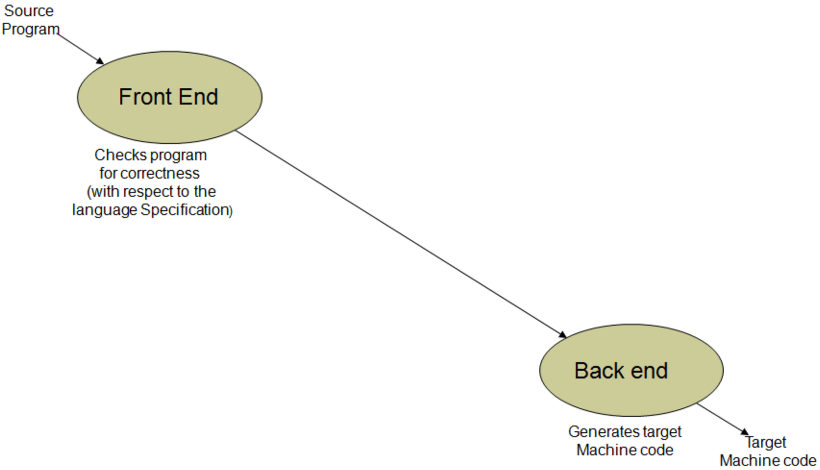
Can compilers become as good as programmers?

Source: Amarsinghe et al

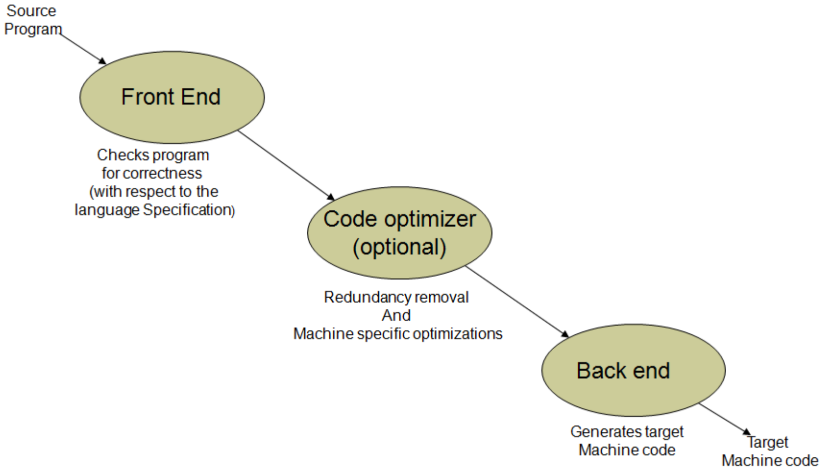
Class of Problems

- Dense data and regular fetch patterns
  - Basic Linear Algebra Systems
  - Use loop optimizations
- Sparse/dense data and irregular fetch patterns
  - N-body simulation, molecular dynamics, Charmm, Discover, Moldyn, Spice, Dyna-3D, Pronto-3D, Gaussian, Dmol, Fidap
  - Inspector-executor model for parallelization
  - Speculative parallelization

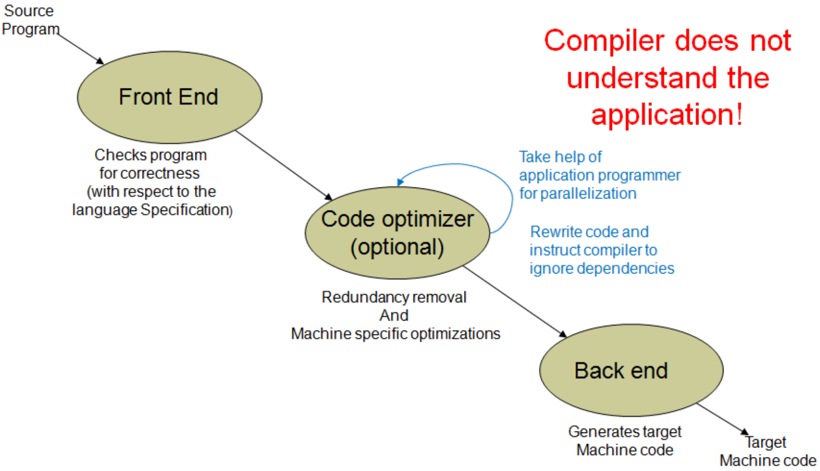
Compiler Structure



Compiler Structure



Compiler Structure



Matrix Multiplication: version 1

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

Matrix Multiplication: version 1

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

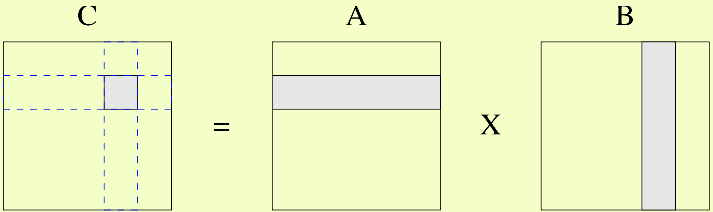
- n^3 iteration
- Each cell computation requires n multiplications and n additions
- Total n^3 multiplications and additions
- Sequential execution time:
  $\sim$ 80 seconds
  (for n=1500 on a dual core laptop)

Matrix Multiplication: version 1

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

- n^3 iteration
- Each cell computation requires n multiplications and n additions
- Total n^3 multiplications and additions
- Sequential execution time:
  $\sim$ 80 seconds
  (for n=1500 on a dual core laptop)

Data Access Pattern



Matrix Multiplication: version 2

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

Matrix Multiplication: version 2

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

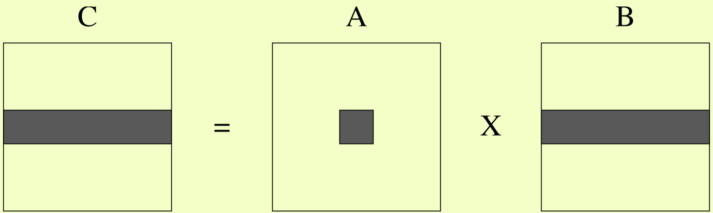
- n^3 iteration
- Each row computation requires n^2 multiplications and n^2 additions
- Total n^3 multiplications and additions
- Sequential execution time:
  $\sim$ 26 seconds
  (for n=1500 on a dual core laptop)

Matrix Multiplication: version 2

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

- n^3 iteration
- Each row computation requires n^2 multiplications and n^2 additions
- Total n^3 multiplications and additions
- Sequential execution time:
  $\sim$ 26 seconds
  (for n=1500 on a dual core laptop)

Data Access Pattern

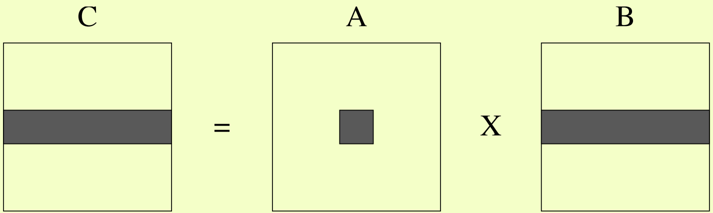


Matrix Multiplication: version 2

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

- n^3 iteration
- Each row computation requires n^2 multiplications and n^2 additions
- Total n^3 multiplications and additions
- Sequential execution time: $\sim$ 26 seconds (for n=1500 on a dual core laptop)

Data Access Pattern



How do we know that version 2 computes the same result as version 1?

Matrix Multiplication: version 3

```
omp_set_num_threads(omp_get_num_procs());
#pragma omp parallel for private(j,k)
for i = 1 to n do
    for j = 1 to n do
        for k = 1 to n do
            c[i,j] += a[i,k] * b[k,j]
        endfor
    endfor
endfor
```

Matrix Multiplication: version 3

```
omp_set_num_threads(omp_get_num_procs());
#pragma omp parallel for private(j,k)
for i = 1 to n do
    for j = 1 to n do
        for k = 1 to n do
            c[i,j] += a[i,k] * b[k,j]
        endfor
    endfor
endfor
```

- n^3 iteration
- Each cell computation requires n multiplications and n additions
- Total n^3 multiplications and additions
- Parallel execution time:
  $\sim$ 50 seconds
  (for n=1500 on a dual core laptop)
- Uses more than one processor!

Matrix Multiplication: version 3

```
omp_set_num_threads(omp_get_num_procs());
#pragma omp parallel for private(j,k)
for i = 1 to n do
    for j = 1 to n do
        for k = 1 to n do
            c[i,j] += a[i,k] * b[k,j]
        endfor
    endfor
endfor
```

- n^3 iteration
- Each cell computation requires n multiplications and n additions
- Total n^3 multiplications and additions
- Parallel execution time: $\sim$ 50 seconds (for n=1500 on a dual core laptop)
- Uses more than one processor!

- How do we know that this version computes the same result as version 1?
- Compilers can easily transform version 1 of the program into version 2 and version 3 to improve performance
- However, they need to do Data Dependence Analysis to establish equivalence of the two versions

Data Dependence Analysis: Example

```
for i = 1, n
    a[i] = b[i]            S1
    c[i] = a[i] + b[i]     S2
    e[i] = c[i+1]          S3
    a[i] = i * i           S4
endfor
```

- S1 writes into a[i] which is read by S2 in the same iteration
- S3 reads from c[i+1] which is over written by S2 in the next iteration
- S1 writes into a[i] which is over written by S4 in the same iteration

Data Dependence Analysis: Example

```
for i = 1, n
    a[i] = b[i]               S1
    c[i] = a[i] + b[i]        S2
    e[i] = c[i+1]             S3
    a[i] = i * i              S4
endfor
```

- S1 writes into a[i] which is read by S2 in the same iteration
- S3 reads from c[i+1] which is over written by S2 in the next iteration
- S1 writes into a[i] which is over written by S4 in the same iteration

- Flow dependence: When a variable is assigned value in one statement and used in a subsequent statement
- Anti dependence: When a variable is used in one statement and reassigned in a subsequent statement
- Output dependence: When a variable is assigned in one statement and reassigned in a subsequent statement

Data Dependence Analysis: Example

```
for i = 1, n
    a[i] = b[i]             S1
    c[i] = a[i] + b[i]      S2
    e[i] = c[i+1]           S3
    a[i] = i * i            S4
endfor
```

- S1 writes into a[i] which is read by S2 in the same iteration
- S3 reads from c[i+1] which is over written by S2 in the next iteration
- S1 writes into a[i] which is over written by S4 in the same iteration

- Flow dependence: When a variable is assigned value in one statement and used in a subsequent statement
- Anti dependence: When a variable is used in one statement and reassigned in a subsequent statement
- Output dependence: When a variable is assigned in one statement and reassigned in a subsequent statement
- Presence of dependence between two statements prevents optimization

- Consider a loop

```
for I = 1, n
    X[ f(I) ] = .....            S1
    ...... = X[ g(I) ]           S2
endfor
```

- There is a dependence from S1 to S2 if there are instances I_1 and I_2 of I such that

$$1 \leq I_1 \leq I_2 \leq N \qquad \text{and} \qquad f(I_1) = g(I_2)$$

there is an iteration I_1 in which S1 writes into X, and a subsequent (or the same) iteration I_2 in which S2 reads from the same element of X

Data Dependence Analysis

- If f and g are general functions, then the problem is intractable.
- If f and g are linear functions of loop indices then to test dependence we need to find values of two integers I_1 and I_2 such that

$$1 \leq I_1 \leq I_2 \leq N \quad \text{and} \quad a_0 + a_1 I_1 = b_0 + b_1 I_2$$

or

$$1 \leq I_1 \leq I_2 \leq N \quad \text{and} \quad a_1 I_1 - b_1 I_2 = b_0 - a_0$$

- These are called Linear Diophantine Equations
- The equations have to be solved to do program optimization

Data Dependence Analysis

- If f and g are general functions, then the problem is intractable.
- If f and g are linear functions of loop indices then to test dependence we need to find values of two integers I_1 and I_2 such that

$$1 \leq I_1 \leq I_2 \leq N \qquad \text{and} \qquad a_0 + a_1 I_1 = b_0 + b_1 I_2$$

or

$$1 \leq I_1 \leq I_2 \leq N \qquad \text{and} \qquad a_1 I_1 - b_1 I_2 = b_0 - a_0$$

- These are called Linear Diophantine Equations
- The equations have to be solved to do program optimization
- IS THAT ALL?

Techniques of Data Dependence Analysis

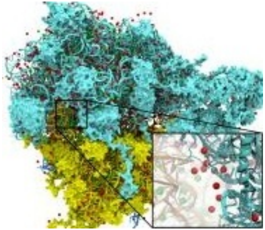
- Reduces to integer programming problem
  - NP Complete
- Exhaustive solutions can not be found
  - iteration space is just too large
- Test equations and inequalities for existence of a solution
- Techniques
  - GCD test for existence of an integer solution
    - could be outside the range
  - Banerjee's test for existence of a solution in the range
    - could be a real solution
  - Omega test: the most powerful test based on Fourier-Motzkin method

Limitations of Data Dependence Analysis

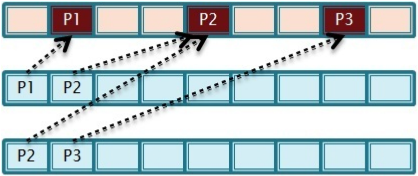
- Can not work with in-exact data and symbolic data
  - Data may not be available at compile time
- Loop iteration count may not be known at compile time
- The iteration space may not be 'well shaped'
- Data access patterns may not be regular (may be very complex!)
- Runtime optimization techniques are required
  - Inspector-Executor model
  - Speculative parallelization

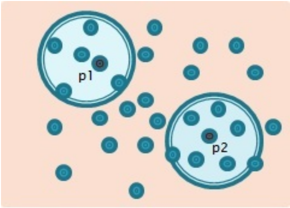
Irregular Access (MOLDYN Kernel)

```
for step = 1, HSTEP
    for i = 1, num_interactions
        n1 = left[i]
        n2 = right[i]
        force = (input[n1] - input[n2])/4
        forces[n1] = forces[n1] + force
        forces[n2] = forces[n2] - force
    endfor
endfor
```



Source: http://gpgpu.org/tag/molecular-dynamics





Inspector Executor Code

```
for iteration = 1 to n
    for i = StartIndex to EndIndex
        a[i] = b[i] + c[ia[i]]
    endfor
endfor
```

Inspector Executor Code

```
for iteration = 1 to n
    for i = StartIndex to EndIndex
        a[i] = b[i] + c[ia[i]]
    endfor
endfor
```

```
//inspector phase
for i = StartIndex to EndIndex
    a[i] = b[i] + c.inspect(ia[i])
endfor

//create the communication schedule
c.schedule()

for iteration = 1 to n
//fetch remote values according
//to the communication schedule
    c.fetch()
    for i = StartIndex to EndIndex
        a[i] = b[i] + c.execute(ia[i])
    endfor
endfor
```

Speculative Parallelization

- Execute loop as a parallel loop
- Keep track of memory references during execution
- Test for data dependence
- If there are dependencies then re-execute the loop sequentially
- LRPD (Lazy Privatizing Doall extended for Reduction validation)
- Improved LRPD with fewer roll backs

Partially Parallel Loop: Example

```
for i = 1, 8
    z = A[K[i]]
    A[L[i]] = z + C[i]
endfor
```

K[1:8] = [1,2,3,1,4,2,1,1]
L[1:8] = [4,5,5,4,3,5,3,3]

iter	1	2	3	4	5	6	7	8
A[]								
1	R			R			R	R
2		R				R		
3			R		W		W	W
4	W			W	R			
5		W	W			W		

Partially Parallel Loop: Example

```
for i = 1, 8
    z = A[K[i]]
    A[L[i]] = z + C[i]
endfor
```

K[1:8] = [1,2,3,1,4,2,1,1]
L[1:8] = [4,5,5,4,3,5,3,3]

iter	1	2	3	4	5	6	7	8
A[]								
1	R			R			R	R
2		R				R		
3			R		W		W	W
4	W			W	R			
5		W	W			W		

- Iterations before the first data dependence are correct and committed.
- Re-apply the LRPD test on the remaining iterations.

Source: Rauchwerger et al

OpenMP (Open Multi-Processing)

- An application programming interface (API)
- Supports multi-platform shared memory multiprocessing programming
- C/C++ and Fortran on many architectures, including Unix and Microsoft Windows platforms.
- Consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior
- Reference: www.openmp.org

State of the Art

- Programmers use high level languages and APIs like OpenMP, Pthreads, Window threads, Mutex, Cuda etc. to write parallel programs
  - GPUs are becoming main stream processors for HPC
- The programmer must have a deep knowledge of concurrency to program these machines
- We are reaching (have already reached?) an era where programmers can not write effective parallel programs without understanding machines and concurrency
- Research compilers have become powerful
  - can achieve performance close to hand coded parallel programs
  - Input of programmer is critical to the compiler performance

The First Compiler

- Fortran (Formula Translation) compiler project 1954-1957

The First Compiler

- Fortran (Formula Translation) compiler project 1954-1957
  - Considered to be one of the ten most influential developments in the history of computing

The First Compiler

- Fortran (Formula Translation) compiler project 1954-1957
  - Considered to be one of the ten most influential developments in the history of computing
  - Nobody believed John Backus when he started the project!!

The First Compiler

- Fortran (Formula Translation) compiler project 1954-1957
  - Considered to be one of the ten most influential developments in the history of computing
  - Nobody believed John Backus when he started the project!!
- Prior to Fortran:
  - Programming was largely done in assembly/machine language
  - Productivity was low

Reasons of Success of Fortran

- End-users were not ignored
  - A mathematical formula could easily be translated into a program
  - Productivity was very high, code maintenance was easy
  - Quality of generated code was very high
- Adoption: about 70-80% programmers were using Fortran within an year
- Side effects: enormous impact on programming languages and computer science
  - Started a new field of research in computer science
  - lead to enormous amount of theoretical work - lexical analysis, parsing, optimization, structured programming, code generation, error recovery etc.

Early Work in Parallelizing Compilers

- Vectorizing compilers (1970s)
- Researchers at UIUC and Rice university have done pioneering work starting in 80s
- First landmark paper appeared in 1987 in TOPLAS
- High quality compilers are available from PGI, Intel, Fujitsu etc.
- Research Compilers are far ahead of production quality compilers

Early Work in Parallelizing Compilers

- Vectorizing compilers (1970s)
- Researchers at UIUC and Rice university have done pioneering work starting in 80s
- First landmark paper appeared in 1987 in TOPLAS
- High quality compilers are available from PGI, Intel, Fujitsu etc.
- Research Compilers are far ahead of production quality compilers

Grand Challenge Problem: Can Fortran experiment be repeated for Parallelizing compilers?

Thank you for your attention

Questions?