Program Analysis & Transformation

**Motivation**
- Map high-level algorithm to low-level architecture
- Improve performance

```
Algorithm
```

```
Architecture
```

```
computation, data, values, code
```

```
instructions, registers, cache, TLB, memory, network, I/O
```

X = ...  
... = Y

parallelism  
locality
Analysis & Transformation – Approaches

- **Automatic**
  - Compiler directed
  - Static / run-time analysis & transformation
  - Low user effort, limited effectiveness

- **Interactive**
  - Programming environment / tool based
  - Display static analysis, apply transformations as directed
  - Moderate user effort, moderately effective

- **Manual**
  - Limited analyses from programming / profiling tools
  - Apply transformations by hand
  - High user effort & effectiveness

Analysis – Dataflow & Dependence

- **Dataflow analysis**
  - Examine flow of values at compile-time
  - Determines control flow & possible variable values
  - Example
    - if (...) { X = 1 } else { X = 2 } Y = X
    - Value of Y is either 1 or 2

- **Data dependence analysis**
  - Examine memory accesses at compile-time
  - Determines locality & constraints on execution order
  - Example
    - Accesses same memory location as 2 iterations earlier
Dependence – Data & Control

- **Data dependences**
  - True / flow \( x = \ldots ; \ldots = x \); read after write
  - Anti \( \ldots = x \); \( x = \ldots \); write after read
  - Output \( x = \ldots ; \ldots = x \); write after write
  - Input \( \ldots = x \); \( \ldots = x \); read after read

- **Control dependence**
  - if (A) { B; } whether B executes depends on result of if

- **Dependence A \( \rightarrow \) B**
  - Dependence from A to B
  - B depends on A
  - A must be executed before B

Dependence – Loop Carried & Independent

- **Loop-carried dependences**
  - Dependence crosses loop iterations
  - Example
    - Dependence occurs across 2 loop iterations

- **Loop-independent dependences**
  - Dependence occurs only on same loop iteration
  - Example
    - Dependence occurs in same loop iteration
Dependence – Parallelism

- **Parallelism & dependence**
  - Computations may be executed in parallel if no dependences
    - Loops may be parallelized if no loop-carried dependences
  - Else data race (result depends on order) may cause error
    - Some exceptions (e.g., input dependence, reduction)

![Diagram of parallel and sequential execution]

Program Transformations

- **Transformations**
  - Change structure of program
  - Improve program in some manner (computation, data)
  - Preserve program output

- **Loop transformations**
  - Change loop structure, iteration order

<table>
<thead>
<tr>
<th>Loop interchange</th>
<th>Loop fission / fusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (X)</td>
<td>for (Y)</td>
</tr>
<tr>
<td>for (Y)</td>
<td>for (X)</td>
</tr>
</tbody>
</table>

... ... A ; B

A

B
Program Transformations

◆ Transformations & dependence
  - Computations may be reordered if dependences preserved
    ● Directly (e.g., instruction scheduling)
    ● Indirectly (e.g., program transformations)
  - Computations may be eliminated if results unused
  - Memory storage can be rearranged

◆ Applying program transformations
  - Ensure output preserved
    ● Preserve dependences (rough approximation)
    ● Preserve dataflow (more precise constraints)
  - Use dependences to guide transformations

Program Transformations

◆ Motivation for transformations
  - Directly improve performance
    ● Increase locality
    ● Exploit parallelism
    ● Etc…
  - Indirectly increase parallelism, enable other transformations
    ● Privatization
    ● Expansion
    ● Reductions
    ● Auxiliary induction variable substitution
    ● Etc…
Privatization & Expansion

◆ Memory-related dependences
  - Caused by reusing memory
  - Can be eliminated by using new memory instead
    ▪ Anti dependence \( \ldots = x \ ; \ x = \ldots \) vs. \( \ldots = x \ ; \ y = \ldots \)
    ▪ Output dependence \( x = \ldots \ ; \ x = \ldots \) vs. \( x = \ldots \ ; \ x = \ldots \)

◆ Approaches
  - Privatization – new memory per processor
  - Expansion – new memory per loop iteration

<table>
<thead>
<tr>
<th>Original</th>
<th>Privatization</th>
<th>Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>do i = int k k = ... ... = k</td>
<td>do i = private int k k = ... ... = k</td>
<td>do i = int k[n] k[i] = ... ... = k[i]</td>
</tr>
</tbody>
</table>

Reductions & Induction Variables

◆ Reductions
  - Associative & commutative operations
    ▪ Sum, multiply, maximum, minimum, etc...
    ▪ Example: \( S = S + A[i] \)
  - Can be executed in any order
    1. Perform reduction on private variable
    2. Combine results to global variable
  - May affect numerical stability for floating point operations

◆ Auxiliary induction variable substitution
  - Variables incremented by fixed amount each loop iteration
    ▪ Example: \( \) for \( i \) \( \) \{ \( k = k + 1; \ p = p + 4; \) \}\)
    ▪ May calculate directly from loop index & eliminate dependence
    ▪ Example: \( \) for \( i \) \( \) \{ \( k = i + c ; \ p = i \times 4; \) \}
Parallelism Optimizations – Synchronization

**Approach**
- Increase size of parallel regions
- Reduce synchronization overhead / load imbalance
- Parallelize outer loops in loop nest
- Merge nearby parallel loops

![Diagram showing loop 1, loop 2 merging into a fused loop]

Parallelism Optimizations – Communications

**Approach**
- Merge smaller messages into large message
- Reduce communication overhead
- Can move communication to deepest loop-carried dependence

```latex
\text{DO } j = m, 1, 1, -1 \\
\text{DO } i = 1, n \\
RX(i,j) = \\
\quad \delta i \quad \text{comm (RX[1:n,j])} \\
\text{DO } i = 1, n \\
RX(i,j) = \\
\quad \text{... RX(i,j+1)}
```
Locality Optimizations

- **Locality**
  - Multiple references to same / nearby locations

- **Types of locality**
  - **Temporal** (reuse data)
  - **Spatial** (reuse nearby data)

![Diagram showing locality concepts]

Processor vs. Memory Speed (Latency)

- Processor Clock
- Memory Bus Clock

<table>
<thead>
<tr>
<th>Year</th>
<th>Speed (Mhz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1988</td>
<td>x86</td>
</tr>
<tr>
<td>1989</td>
<td>x86</td>
</tr>
<tr>
<td>1990</td>
<td>x86</td>
</tr>
<tr>
<td>1991</td>
<td>x86</td>
</tr>
<tr>
<td>1992</td>
<td>x86</td>
</tr>
<tr>
<td>1993</td>
<td>x86</td>
</tr>
<tr>
<td>1994</td>
<td>x86</td>
</tr>
<tr>
<td>1995</td>
<td>x86</td>
</tr>
<tr>
<td>1996</td>
<td>x86</td>
</tr>
<tr>
<td>1997</td>
<td>x86</td>
</tr>
<tr>
<td>1998</td>
<td>x86</td>
</tr>
<tr>
<td>1999</td>
<td>x86</td>
</tr>
<tr>
<td>2000</td>
<td>x86</td>
</tr>
<tr>
<td>2001</td>
<td>x86</td>
</tr>
<tr>
<td>2002</td>
<td>x86</td>
</tr>
</tbody>
</table>

- FPM DRAM (420 ns)
- EDO DRAM (300 ns)
- SDRAM (200 ns)
- DDR-DRAM (200 ns)
Regular Memory Access Patterns

**Characteristics**
- Multidimensional arrays
- Multiple loop nests
- Also image processing, database scans

**Goal**
- Unit-stride access $\rightarrow$ exploit spatial locality

Regular codes

```plaintext
do i = 1, N
  do j = 1, N
    ... = node[j, i]
```

Program Transformations – Tiling

**Approach**
- Move reuses closer in time
- Better use of processor cache

```plaintext
do J=1,N
  do K=1,N
    do I=1,N
```

- Tile data should now fit in cache

```plaintext
do KK=1,N,TK
  do II=1,N,TI
    do J=1,N
      do K=KK,min(KK+TK-1,N)
        do I=II,min(II+TI-1,N)
```
Irregular Memory Access Patterns

◆ Characteristics
- Memory accesses via index array or pointers
- Irregular memory accesses ⇒ poor locality
- Requires run-time transformations

◆ Goal
- Reorder data / accesses → exploit temporal / spatial locality

Irregular codes

do i = 1, M
... = node[ edge1[i] ]
... = node[ edge2[i] ]

Locality Transformations

◆ Reorder data & computation for cache

◆ Distribute data & computation to processors
Types of Parallel Programming

- **Multiprogramming**
  - Multiple, unrelated, instruction streams
  - Execute on single or multiple processors
  - Overlap execution to hide latency, fully utilize resources
  - Increases throughput (reduce execution time for all programs)
  - Does not reduce execution time of single program

- **Parallel & distributed programming**
  - Multiple, related, interacting instruction streams
  - Execute on multiple processors
  - Incurred overhead, underutilize resources
  - Reduce execution time of single program

- **Parallel computing**
  - Fine-grain, data parallelism
  - Frequent inter-processor communication & synchronization
  - Performance requires hardware support

- **Distributed computing**
  - Coarse-grain, task-level parallelism
  - Infrequent inter-processor communication
    - Mostly at beginning / end of computation
  - Little hardware support required
  - Also known as “embarrassingly parallel”
Program Performance – Communication

- **Communication / computation ratio**
  - Constraint on parallel performance
  - High ratio = low performance

![Graph showing speedup vs. number of processors for low and high communication](image)

Program Performance – Data

- **Data access / computation ratio**
  - Constraint on sequential performance
  - High ratio = low performance

![Graph showing performance vs. data size for cache and memory bandwidth](image)
Bioinformatics Applications

✦ Current practice
  - Usually embarrassingly parallel
  - Use either multiprogramming or distributed computing
  - On collection of servers

✦ NCBI example
  - NCBI maintains cluster of 80+ PCs for GenBank
  - Web server receives request to “blast” sequences X, Y, Z...
  - Farms out individual requests to separate PCs
  - Collects answer and creates web page with result

✦ As size of sequence databases grow
  - May need to exploit parallelism for individual applications

Sequence Alignment / Search and HPC

✦ Any need for high performance computing?
  - Maybe

✦ BLAST algorithm
  - Linear scan over flat (ASCII) sequence database
  - Embarrassingly parallel

✦ Current parallel implementations
  - MPI-BLAST, TURBO-BLAST (Linda-based)
    - Speed up individual searches
  - Distributed BLAST
    - BLAST queries assigned to individual PC in Biocluster

✦ Potential research area
  - Parallel high-precision multiple sequence search / alignment
Protein Structure Prediction and HPC

- **Need for high performance computing?**
  - In some cases
- **Ab initio algorithms**
  - Fine-grain parallel, very computationally expensive
- **Comparative modeling algorithms**
  - Fine-grain parallel, currently low-medium computation
- **Threading algorithms**
  - Embarassingly parallel, currently low-medium computation
- **Current parallel implementations**
  - Ab initio methods (molecular dynamics), threading
- **Potential research area**
  - Parallel high-precision comparative modeling

Protein-Ligand Docking and HPC

- **Need for high performance computing?**
  - Maybe
- **Algorithm**
  - Embarassingly parallel
  - Can test each ligand in parallel
- **Potential research area**
  - Parallel high-precision protein-ligand docking analysis
Gene Expression Analysis and HPC

- **Need for high performance computing?**
  - Maybe

- **Algorithm**
  - Data mining large microarray databases
  - Computation depends on level of detail

- **Potential research area**
  - Parallel high-precision cluster analysis

Phylogenetic Analysis and HPC

- **Need for high performance computing?**
  - Yes

- **Algorithm**
  - Embarrassingly parallel
  - Evaluate possible trees in parallel

- **Current parallel implementations**
  - GRAPPA, etc...

- **Potential research area**
  - Parallel high-precision phylogenetic analysis
Bioinformatics and Parallel Computing

- **Targets for high performance computing**
  - Sequence alignment / search  
    embarrassingly parallel
  - Protein structure prediction  
    fine-grain parallel
  - Protein docking  
    embarrassingly parallel
  - Gene expression analysis  
    parallel
  - Phylogenetic analysis  
    embarrassingly parallel

- **Open question**
  - What fields of bioinformatics will benefit…
    …if parallel computing enables more powerful algorithms