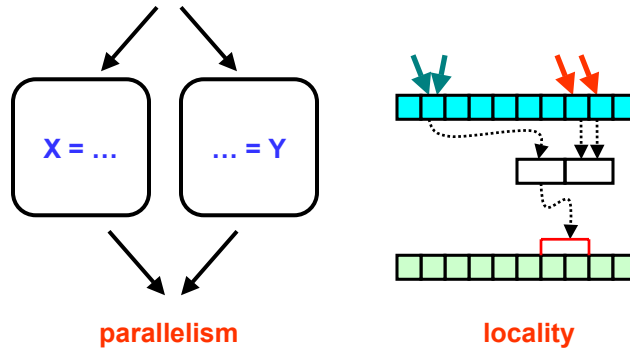


CMSC 838T – Lecture 13

◆ Program analysis & transformation

- Data-flow & data dependence analyses
- Guide program transformations
- Improve parallelism & locality

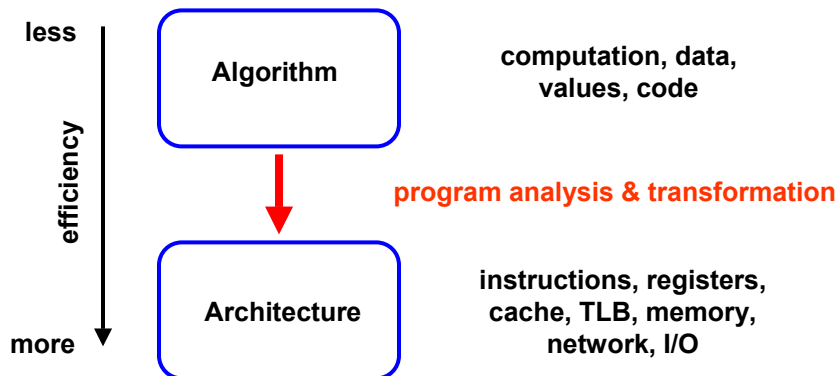


CMSC 838T – Lecture 13

Program Analysis & Transformation

◆ Motivation

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



CMSC 838T – Lecture 13

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

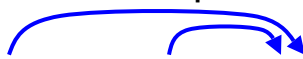
CMSC 838T – Lecture 13

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

-  for (K) { A[K] = A[K - 2] }
- Accesses same memory location as 2 iterations earlier

CMSC 838T – Lecture 13

Dependence – Data & Control

◆ Data dependences

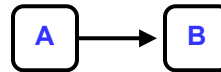
- True / flow $x = \dots ; \dots = x ;$ read after write
- Anti $\dots = x ; x = \dots ;$ write after read
- Output $x = \dots ; x = \dots ;$ write after write
- Input $\dots = x ; \dots = x ;$ read after read

◆ 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




CMSC 838T – Lecture 13

Dependence – Loop Carried & Independent

◆ Loop-carried dependences

- Dependence crosses loop iterations
- Example 
 - for (K) { A[K] = A[K - 2] }
 - Dependence occurs across 2 loop iterations

◆ Loop-independent dependences

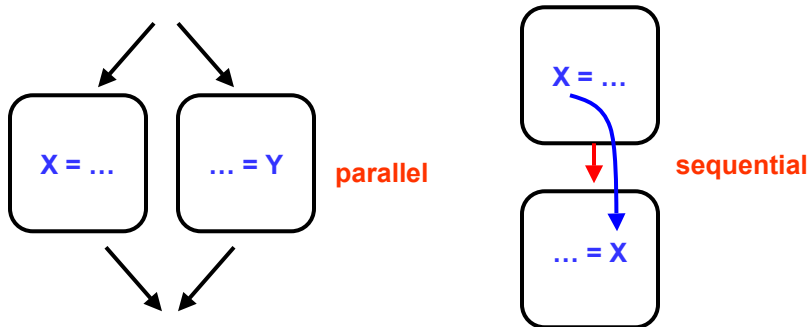
- Dependence occurs only on same loop iteration
- Example 
 - for (K) { A[K] = A[K] + 1 }
 - Dependence occurs in same loop iteration

CMSC 838T – Lecture 13

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)



CMSC 838T – Lecture 13

Program Transformations

◆ Transformations

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

◆ Loop transformations

- Change loop structure, iteration order

Loop interchange

```
for ( X )  
  for ( Y )  
  ...  
↔  
for ( Y )  
  for ( X )  
  ...
```

Loop fission / fusion

```
for ( X )  
  A ; B  
↔  
for ( X )  
  A  
  for ( X )  
  B
```

CMSC 838T – Lecture 13

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

CMSC 838T – Lecture 13

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...

CMSC 838T – Lecture 13

Privatization & Expansion

◆ Memory-related dependences

- Caused by reusing memory
- Can be eliminated by using new memory instead
 - Anti dependence ... = x ; x = ... vs. ... = x ; y = ...
 - Output dependence x = ... ; x = ... vs. x = ... ; x = ...

◆ Approaches

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

Original	Privatization	Expansion
do i =	do i =	do i =
int k	private int k	int k[n]
k = ...	k = ...	k[i] = ...
... = k	... = k	... = k [i]

CMSC 838T – Lecture 13

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

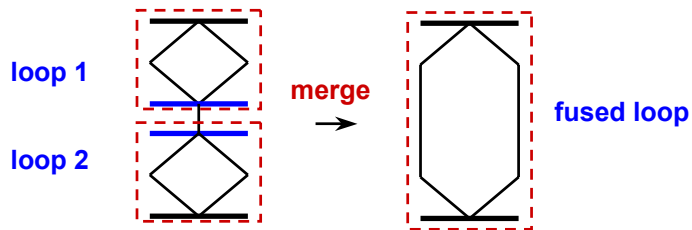
- Variables incremented by fixed amount each loop iteration
 - Example: for (i) { k = k + 1 ; p = p + 4 ; }
- May calculate directly from loop index & eliminate dependence
 - Example: for (i) { k = i + c ; p = i * 4 ; }

CMSC 838T – Lecture 13

Parallelism Optimizations – Synchronization

◆ Approach

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

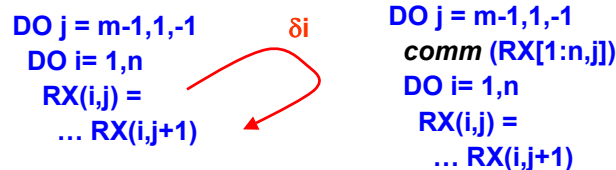


CMSC 838T – Lecture 13

Parallelism Optimizations – Communications

◆ Approach

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



CMSC 838T – Lecture 13

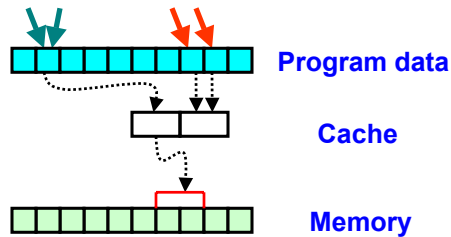
Locality Optimizations

◆ Locality

- Multiple references to same / nearby locations

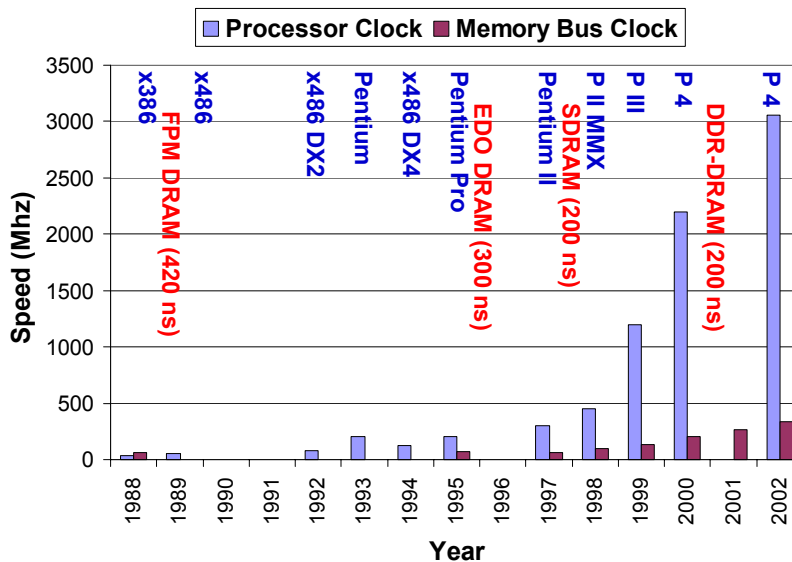
◆ Types of locality

- **Temporal** (reuse data)
- **Spatial** (reuse nearby data)



CMSC 838T – Lecture 13

Processor vs. Memory Speed (Latency)



CMSC 838T – Lecture 13

Regular Memory Access Patterns

◆ Characteristics

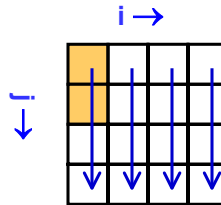
- Multidimensional arrays
- Multiple loop nests
- Also image processing, database scans

◆ Goal

- Unit-stride access → exploit spatial locality

Regular codes

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



CMSC 838T – Lecture 13

Program Transformations – Tiling

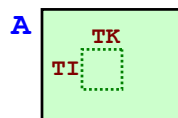
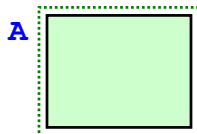
◆ Approach

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

```
do J=1,N
  do K=1,N
    do I=1,N
      C[I,J] = C[I,J] +
        A[I,K] * B[K,J]
```



```
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)
          C[I,J] = C[I,J] +
            A[I,K] * B[K,J]
```



- Tile data should now fit in cache

CMSC 838T – Lecture 13

Irregular Memory Access Patterns

◆ Characteristics

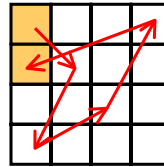
- Memory accesses via **index array** or **pointers**
- Irregular memory accesses \Rightarrow **poor locality**
- Requires **run-time** transformations

◆ Goal

- Reorder data / accesses \rightarrow exploit temporal / spatial locality

Irregular codes

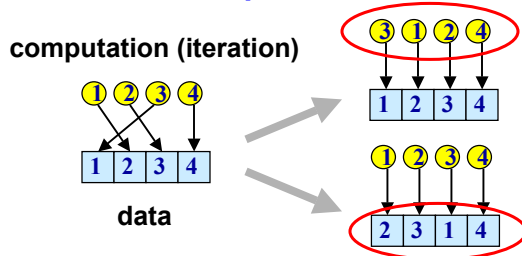
```
do i = 1, M
  ... = node[ edge1[i] ]
  ... = node[ edge2[i] ]
```



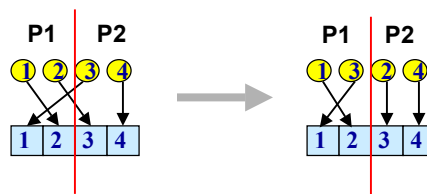
CMSC 838T – Lecture 13

Locality Transformations

◆ Reorder data & computation for cache



◆ Distribute data & computation to processors



CMSC 838T – Lecture 13

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
 - Incur overhead, underutilize resources
- Reduce execution time of **single** program

CMSC 838T – Lecture 13

Types of Parallel Programming

◆ Parallel computing

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

◆ Distributed computing

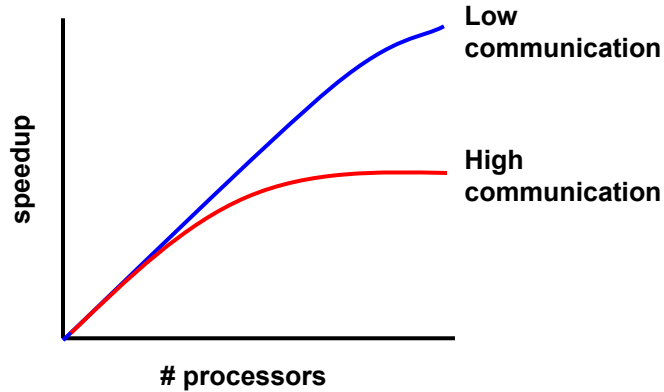
- Coarse-grain, task-level parallelism
- Infrequent inter-processor communication
 - Mostly at beginning / end of computation
- Little hardware support required
- Also known as “embarrassingly parallel”

CMSC 838T – Lecture 13

Program Performance – Communication

◆ Communication / computation ratio

- Constraint on parallel performance
- High ratio = low performance

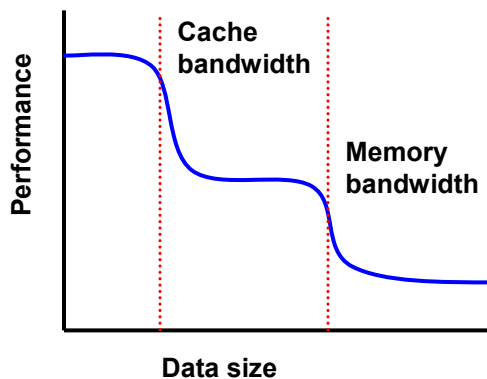


CMSC 838T – Lecture 13

Program Performance – Data

◆ Data access / computation ratio

- Constraint on sequential performance
- High ratio = low performance



CMSC 838T – Lecture 13

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 create web page with result
- ◆ **As size of sequence databases grow**
 - May need to exploit parallelism for individual applications

CMSC 838T – Lecture 13

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

CMSC 838T – Lecture 13

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**
 - Embarrassingly parallel, currently low-medium computation
- ◆ **Current parallel implementations**
 - Ab initio methods (molecular dynamics), threading
- ◆ **Potential research area**
 - Parallel high-precision comparative modeling

CMSC 838T – Lecture 13

Protein-Ligand Docking and HPC

- ◆ **Need for high performance computing?**
 - Maybe
- ◆ **Algorithm**
 - Embarrassingly parallel
 - Can test each ligand in parallel
- ◆ **Potential research area**
 - Parallel high-precision protein-ligand docking analysis

CMSC 838T – Lecture 13

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

CMSC 838T – Lecture 13

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

CMSC 838T – Lecture 13

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