CS4961 Parallel Programming

Lecture 12/13: Introduction to Locality

Mary Hall
October 1/3, 2009

Administrative

- Programming assignment 2 is posted (after class)
- Due, Thursday, October 8 before class
  - Use the "handin" program on the CADE machines
  - Use the following command:

    "handin cs4961 prog2 <gzipped tar file>"
- Mailing list set up: cs4961@list.eng.utah.edu
- Midterm Quiz on Oct. 8
  - Covers material in Lectures 1-11
  - Brief review on Tuesday
  - Can take it early on Tuesday, Oct. 6 after class

Today's Lecture

- Questions on assignment
- Review for exam
- More on data locality (and slides from last time)
  - Begin with dependences
  - Tests for legality of transformations
  - A few more transformations

Review for Quiz

- L1: Overview
  - Technology drivers for multi-core paradigm shift
  - Concerns in efficient parallelization
- L2:
  - Fundamental theorem of dependence (also in today's lecture)
  - Reductions
- L3
  - SIMD/MIMD, shared memory, distributed memory
  - Candidate Type Architecture Model
- L4
  - Task and data parallelism,
  - Example in Peril-L
- L5
  - Task Parallelism and Task Graphs

Review for Quiz

- L6
  - SIMD execution
  - Challenges with SIMD multimedia extensions
- L7
  - Solution to HW2
  - Ghost cells and data partitioning
- L8 & L9
  - OpenMP: constructs, idea, target architecture
- L10
  - TBB (see tutorial and assignment)
- L11
  - Reasoning about Performance

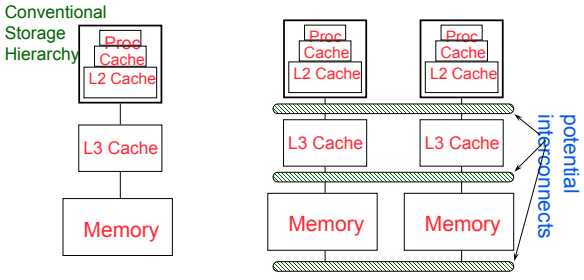
Format for Quiz

- Definitions, 1 from each lecture
- Problem solving (3 questions)
- Essay question (pick one from 5)

Locality – What does it mean?

- We talk about locality a lot
- What are the ways to improve memory behavior in a parallel application?
- What are the key considerations?
- Today's lecture
  - Mostly about managing caches (and registers)
  - Targets of optimizations
  - Abstractions to reason about locality
  - Data dependences, reordering transformations

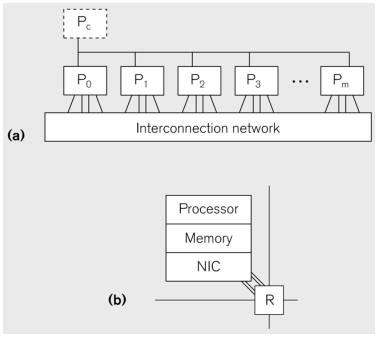
Locality and Parallelism (from Lecture 1)



- Large memories are slow, fast memories are small
- Cache hierarchies are intended to provide illusion of large, fast memory
- Program should do most work on local data!

Lecture 3: Candidate Type Architecture (CTA Model)

- A model with *P* standard processors, *d* degree, λ latency
- Node == processor + memory + NIC
- Key Property: Local memory ref is 1, global memory is λ



Targets of Memory Hierarchy Optimizations

- Reduce ***memory latency***
  - The latency of a memory access is the time (usually in cycles) between a memory request and its completion
- Maximize ***memory bandwidth***
  - Bandwidth is the amount of useful data that can be retrieved over a time interval
- Manage overhead
  - Cost of performing optimization (e.g., copying) should be less than anticipated gain

Reuse and Locality

- Consider how data is accessed
  - ***Data reuse:***
    - Same or nearby data used multiple times
    - Intrinsic in computation
  - ***Data locality:***
    - Data is reused and is present in "fast memory"
    - Same data or same data transfer
- If a computation has reuse, what can we do to get locality?
  - Appropriate data placement and layout
  - Code reordering transformations

Cache basics: a quiz

Parameters to optimization

- **Cache hit:**
  - in-cache memory access—cheap
- **Cache miss:**
  - non-cached memory access—expensive
  - need to access next, slower level of hierarchy
- **Cache line size:**
  - # of bytes loaded together in one entry
  - typically a few machine words per entry
- **Capacity:**
  - amount of data that can be simultaneously in cache
- **Associativity**
  - direct-mapped: only 1 address (line) in a given range in cache
  - *n*-way: $n \geq 2$ lines w/ different addresses can be stored

Temporal Reuse in Sequential Code

- Same data used in distinct iterations I and I'

```
for (i=1; i<N; i++)
  for (j=1; j<N; j++)
    A[j]= A[j+1]+A[j-1]
```

- `A[j]` has self-temporal reuse in loop `i`

Spatial Reuse (Ignore for now)

- Same data transfer (usually cache line) used in distinct iterations I and I'

```
for (i=1; i<N; i++)
  for (j=1; j<N; j++)
    A[j]= A[j+1]+A[j-1];
```

- `A[j]` has self-spatial reuse in loop `j`
- **Multi-dimensional array note:** C arrays are stored in row-major order

Group Reuse

- Same data used by distinct references

```
for (i=1; i<N; i++)
  for (j=1; j<N; j++)
    A[j]= A[j+1]+A[j-1];
```

- `A[j]`,`A[j+1]` and `A[j-1]` have group reuse (spatial and temporal) in loop `j`

Can Use Reordering Transformations!

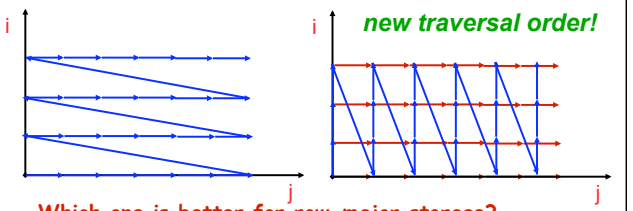
- Analyze reuse in computation
- Apply loop reordering transformations to improve locality based on reuse
- With any loop reordering transformation, always ask
  - **Safety?** (doesn't reverse dependences)
  - **Profitablity?** (improves locality)

Loop Permutation: An Example of a Reordering Transformation

Permute the order of the loops to modify the traversal order

```
for (i= 0; i<3; i++)
 for (j=0; j<6; j++)
   A[i][j+1]=A[i][j]+B[j];
```

```
for (j=0; j<6; j++)
 for (i= 0; i<3; i++)
   A[i][j+1]=A[i][j]+B[j];
```



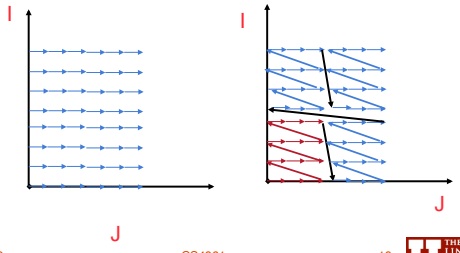
Which one is better for row-major storage?

Permutation has many goals

- Locality optimization
  - Particularly, for spatial locality (like in your SIMD assignment)
- Rearrange loop nest to move parallelism to appropriate level of granularity
  - Inward to exploit fine-grain parallelism (like in your SIMD assignment)
  - Outward to exploit coarse-grain parallelismx
- Also, to enable other optimizations

Tiling (Blocking): Another Loop Reordering Transformation

- Blocking reorders loop iterations to bring iterations that reuse data closer in time
- Goal is to retain in cache between reuse



Tiling is Fundamental!

- Tiling is very commonly used to manage limited storage
  - Registers
  - Caches
  - Software-managed buffers
  - Small main memory
- Can be applied hierarchically

Tiling Example

```
for (j=1; j<M; j++)
   for (i=1; i<N; i++)
     D[i] = D[i] +B[j,i]
```

Strip mine

```
for (j=1; j<M; j++)
   for (i=1; i<N; i+=s)
       for (ii=i; ii<min(i+s-1,N); ii++)
            D[ii] = D[ii] +B[j,ii]
```

Permute

```
for (i=1; i<N; i+=s)
   for (j=1; j<M; j++)
      for (ii=i; ii<min(i+s-1,N); ii++)
           D[ii] = D[ii] +B[j,ii]
```


How to Determine Safety and Profitability?

- Safety
  - A step back to Lecture 2
  - Notion of reordering transformations
  - Based on data dependences
- Profitability
  - Reuse analysis (and other cost models)
  - Also based on data dependences, but simpler

Key Control Concept: Data Dependence

- **Question:** When is parallelization guaranteed to be safe?
- **Answer:** If there are no data dependences across reordered computations.
- **Definition:** Two memory accesses are involved in a data dependence if they may refer to the same memory location and one of the accesses is a write.
- **Bernstein's conditions (1966):** I_j is the set of memory locations read by process P_j, and O_j the set updated by process P_j. To execute P_j and another process P_k in parallel,

$I_j \cap O_k = \phi$ write after read

$I_k \cap O_j = \phi$ read after write

$O_j \cap O_k = \phi$ write after write

Data Dependence and Related Definitions

- Actually, parallelizing compilers must formalize this to guarantee correct code.
- Let's look at how they do it. It will help us understand how to reason about correctness as programmers.
- **Definition:**
  Two memory accesses are involved in a data dependence if they may refer to the same memory location and one of the references is a write.

  A data dependence can either be between two distinct program statements or two different dynamic executions of the same program statement.

- **Source:**
  - "Optimizing Compilers for Modern Architectures: A Dependence-Based Approach", Allen and Kennedy, 2002, Ch. 2. (not required or essential)

Some Definitions (from Allen & Kennedy)

- **Definition 2.5:**
  - Two computations are equivalent if, on the same inputs,
    - they produce identical outputs
    - the outputs are executed in the same order
- **Definition 2.6:**
  - A reordering transformation
    - changes the order of statement execution
    - without adding or deleting any statement executions.
- **Definition 2.7:**
  - A reordering transformation preserves a dependence if
    - it preserves the relative execution order of the dependences' source and sink.

Fundamental Theorem of Dependence

- **Theorem 2.2:**
  - Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.

Brief Detour on Parallelizable Loops as a Reordering Transformation

Forall or Doall loops:
Loops whose iterations can execute in parallel (a particular reordering transformation)

Example

```
forall (i=1; i<=n; i++)
    A[i] = B[i] + C[i];
```

Meaning?

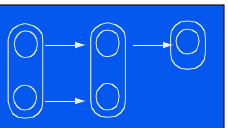
Each iteration can execute independently of others
Free to schedule iterations in any order

Why are parallelizable loops an important concept?

Source of scalable, balanced work
Common to scientific, multimedia, graphics & other domains

Data Dependence for Arrays

```
for (i=2; i<5; i++)
    A[i] = A[i-2]+1;
```



Loop-Carried dependence

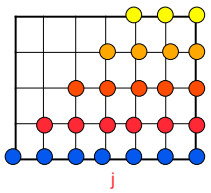
```
for (i=1; i<=3; i++)
    A[i] = A[i]+1;
```

Loop-Independent dependence

- Recognizing parallel loops (intuitively)
  - Find data dependences in loop
  - No dependences crossing iteration boundary ➔ parallelization of loop's iterations is safe

1. Characterize Iteration Space

```
for (i=1;i<=5; i++)
  for (j=i;j<=7; j++)
     ...
```

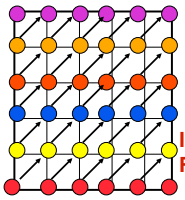


$1 \le i \le 5$
$i \le j \le 7$

- **Iteration instance:** represented as coordinates in iteration space
  - *n*-dimensional discrete cartesian space for *n* deep loop nests
- **Lexicographic order:** Sequential execution order of iterations
  [1,1], [1,2], …, [1,6],[1,7],
  [2,2], [2,3], …, [2,6], …
- Iteration I (a vector) is lexicographically less than I', I<I' , iff there exists c (i_1, …, i_{c-1}) = (i'_1, …, i'_{c-1}) and $i_c < i'_c$.

2. Compare Memory Accesses across Dynamic Instances in Iteration Space

```
N = 6;
for (i=1; i<N; i++)
   for (j=1; j<N; j++)
      A[i+1,j+1] = A[i,j] * 2.0;
```



I'=[2,2], Read A[2,2

I=[1,1], Write A[2,2]

How to describe relationship between two dynamic instances?
e.g., **I=[1,1]** and **I'=[2,2]**

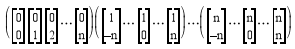
Distance Vectors

```
N = 6;
for (i=1; i<N; i++)
   for (j=1; j<N; j++)
      A[i+1,j+1] = A[i,j] * 2.0;
```

- Distance vector = **[1,1]**
- A loop has a distance vector D if there exists data dependence from iteration vector **I** to a later vector **I'**, and **D = I' - I**.
- Since I' > I, **D >= 0**.
  (D is lexicographically greater than or equal to 0).

Distance and Direction Vectors

- Distance vectors: (infinitely large set)



- Direction vectors: (realizable if 0 or lexicographically positive)
  ([=,=],[=,<],[<,>], [<,=], [<,<])
- Common notation:

0	=
+	<
-	>
+/-	*

Parallelization Test: 1-Dimensional Loop

- **Examples:**

```
for (j=1; j<N; j++)                for (j=1; j<N; j++)
    A[j] = A[j] + 1;                   B[j] = B[j-1] + 1;
```

- **Dependence (Distance and Direction) Vectors?**
- **Test for parallelization:**
  - A 1-d loop is parallelizable if for all data dependences D ε **D**, D = 0

n-Dimensional Loop Nests

```
for (i=1; i<=N; i++)
   for (j=1; j<=N; j++)
      A[i,j] = A[i,j-1]+1;

for (i=1; i<=N; i++)
   for (j=1; j<=N; j++)
      A[i,j] = A[i-1,j-1]+1;
```

- **Distance and direction vectors?**
- **Definition:**
  $D = (d_1, \ldots d_n)$ is loop-carried at level i if d_i is the first nonzero element.

Test for Parallelization

The i th loop of an n-dimensional loop is parallelizable if there does not exist any level i data dependences.

The i th loop is parallelizable if for all dependences $D = (d_1, \ldots d_n)$,
either
$(d_1, \ldots d_{i-1}) > 0$
or
$(d_1, \ldots d_i) = 0$

Back to Locality: Safety of Permutation

- **Intuition:** Cannot permute two loops i and j in a loop nest if doing so reverses the direction of any dependence.
- Loops i through j of an n-deep loop nest are *fully permutable* if for all dependences D,
  either
  $(d_1, \ldots d_{i-1}) > 0$
  or
  forall k, $i \le k \le j$, $d_k \ge 0$
- **Stated without proof:** Within the affine domain, n-1 inner loops of n-deep loop nest can be transformed to be fully permutable.

Simple Examples: 2-d Loop Nests

```
for (i= 0; i<3; i++)
   for (j=0; j<6; j++)
     A[i][j+1]=A[i][j]+B[j]
```

```
for (i= 0; i<3; i++)
   for (j=0; j<6; j++)
     A[i+1][j-1]=A[i][j]
                 +B[j]
```

- Distance vectors
- Ok to permute?

Legality of Tiling

- Tiling = strip-mine and permutation
  - -Strip-mine does not reorder iterations
  - -Permutation must be legal

  OR

  - strip size less than dependence distance

Summary of Lecture

- Motivation for Locality Optimization
- Discussion of dependences and reuse analysis
- Brief examination of loop transformations
  - Specifically, permutation and tiling
  - More next lecture