CS4961 Parallel Programming

Lecture 7: More Dependences and Introduction to Locality

Mary Hall
September 13, 2011

Administrative

- Nikhil's office hours are moved to Wednesday this week
  - 2-3PM, MEB3115, Desk#12
- I will be on travel a week from today, but there will be class. More on Thursday.

Programming Assignment 1: Due Wednesday, Sept. 21, 11:59PM

To be done on water.eng.utah.edu (you all have accounts – passwords available if your CS account doesn't work)

1. Write an average of a set of numbers in OpenMP for a problem size and data set to be provided. Use a block data distribution.
2. Write the same computation in Pthreads.

Report your results in a separate README file.

- What is the parallel speedup of your code? To compute parallel speedup, you will need to time the execution of both the sequential and parallel code, and report speedup = Time(seq) / Time (parallel)
- If your code does not speed up, you will need to adjust the parallelism granularity, the amount of work each processor does between synchronization points.
- Report results for two different numbers of threads.

Extra credit: Rewrite both codes using a cyclic distribution

Programming Assignment 1, cont.

- A test harness is provided in avg-test-harness.c that provides a sequential average, validation, speedup timing and substantial instructions on what you need to do to complete the assignment.
- Here are the key points:
  - You'll need to write the parallel code, and the things needed to support that. Read the top of the file, and search for "TODO".
  - Compile w/ OpenMP: cc –o avg-openmp –O3 –xopenmp avg-openmp.c
  - Compile w/ Pthreads:

    cc –o avg-pthreads –O3 avg-pthreads.c –lpthread
  - Run OpenMP version: ./avg-openmp > openmp.out
  - Run Pthreads version: ./avg-pthreads > pthreads.out
- Note that editing on water is somewhat primitive – I'm using vim. You may want to edit on a different machine and copy to water, but keep in mind that you'll need a fast edit-compile-execute path. Or you can try vim, too. ☺

Today's Lecture

- Data Dependences
  - How compilers reason about them
  - Formal definition of reordering transformations that preserve program meaning
  - Informal determination of parallelization safety
- Locality
  - Data reuse vs. data locality
  - Introduction to reordering transformations for locality
- Sources for this lecture:
  - Notes on website

Race Condition or Data Dependence

- A ***race condition*** exists when the result of an execution depends on the ***timing*** of two or more events.
- A ***data dependence*** is an ordering on a pair of memory operations that must be preserved to maintain correctness.

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.

Data Dependence of Scalar Variables

True (flow) dependence

```
a      =
     = a
```

Anti-dependence

```
     = a
a      =
```

Output dependence

```
a      =
a      =
```

Input dependence (for locality)

```
= a

= a
```

Definition: Data dependence exists from a reference instance i to i' iff

either i or i' is a write operation
i and i' refer to the same variable
i executes before i'

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.

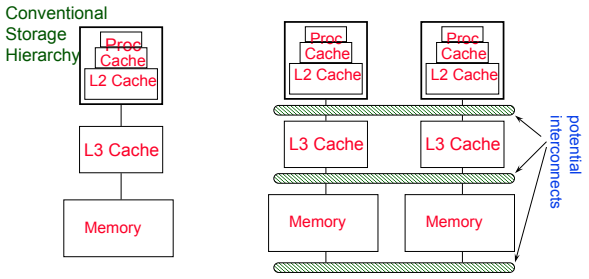
Fundamental Theorem of Dependence

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

In this course, we consider two kinds of reordering transformations

- Parallelization
  - Computations that execute in parallel between synchronization points are potentially reordered. Is that reordering safe? According to our definition, it is safe if it preserves the dependences in the code.
- Locality optimizations
  - Suppose we want to modify the order in which a computation accesses memory so that it is more likely to be in cache. This is also a reordering transformation, and it is safe if it preserves the dependences in the code.
- Reduction computations
  - We have to relax this rule for reductions. It is safe to reorder reductions for commutative and associative operations.

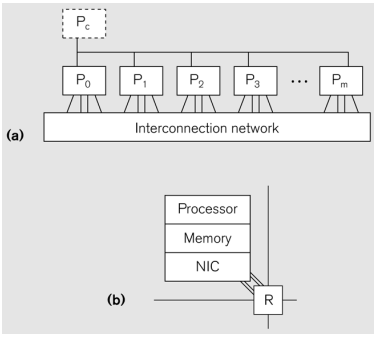
Locality and Parallelism (from Lecture 1)



- 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*



Managing Locality

- Mostly, we have focused on accessing data used by a processor from local memory
  - We call this data partitioning or data placement
  - Let's take a look at this Red/Blue example
- But we can also manage locality within a processor in its cache and registers
  - We'll look at this too!
  - Not really a parallel programming problem, but if you do not think about locality, you may give up a lot of performance.

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

- **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

Parameters to optimization

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

- 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`
- For multi-dimensional arrays, depends on how array is stored.

Exploiting Reuse: Locality optimizations

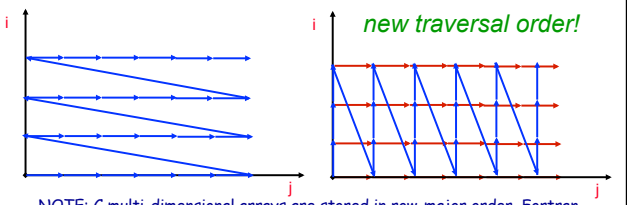
- We will study a few loop transformations that reorder memory accesses to improve locality.
- These transformations are also useful for parallelization too (to be discussed later).
- Two key questions:
  - Safety:
    - Does the transformation preserve dependences?
  - Profitability:
    - Is the transformation likely to be profitable?
    - Will the gain be greater than the overheads (if any) associated with the transformation?

Loop Transformations: Loop Permutation

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];
```



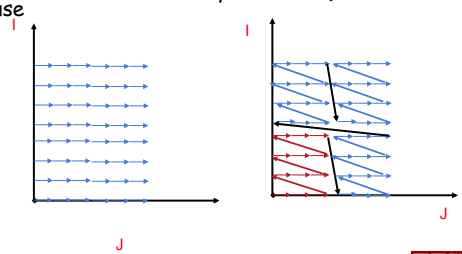
NOTE: C multi-dimensional arrays are stored in row-major order, Fortran in column major

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
  - Outward to exploit coarse-grain parallelism
- 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/register/scratchpad (or other constrained memory structure) 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
- Also used in context of managing granularity of parallelism

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

Permute

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


A More Formal Treatment of Safety

- We'll develop the dependence concept
- And provide some abstractions
  - Distance vectors