CS4961 Parallel Programming

Lecture 8: Dependences and Locality Optimizations

Mary Hall
September 15, 2011

Administrative

- I will be on travel Tuesday, September 20
- Nikhil will hold lab hours to complete the programming assignment
  - ROOM L130

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. Apparently, you can edit on CADE machines and just run on water. Or you can try vim, too. ☺

Today's Lecture

- Data Dependences
  - How compilers reason about them
  - Informal determination of parallelization safety
- Locality
  - Data reuse vs. data locality
  - Reordering transformations for locality
- Sources for this lecture:
  - Notes on website

Data Dependence and Related Definitions

- **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.

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.

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

Exploiting Reuse: Locality optimizations

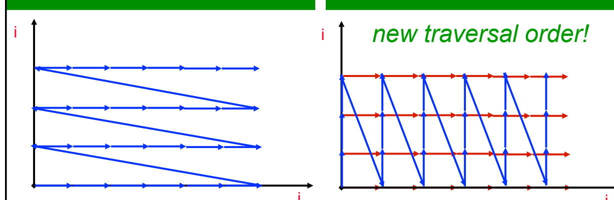
- We will study a few loop transformations that reorder memory accesses to improve locality.
- These transformations are also useful for parallelization too (to be discussed later).
- Two key questions:
  - Safety:
    - Does the transformation preserve dependences?
  - Profitability:
    - Is the transformation 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];
```

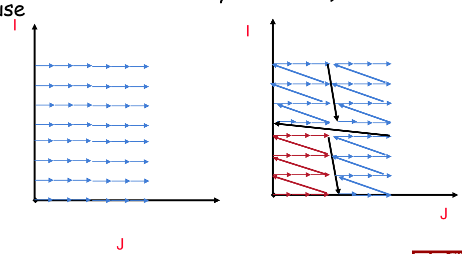
```
for (j=0; j<6; j++)
 for (i= 0; i<3; i++)
   A[i][j+1]=A[i][j]+B[j];
```



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

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 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]
```


Unroll, Unroll-and-Jam

- Unroll simply replicates the statements in a loop, with the number of copies called the unroll factor
- As long as the copies don't go past the iterations in the original loop, it is always safe
  - May require "cleanup" code
- Unroll-and-jam involves unrolling an outer loop and fusing together the copies of the inner loop (not always safe)
- One of the most effective optimizations there is, but there is a danger in unrolling too much

```
Original:
for (i=0; i<4; i++)
 for (j=0; j<8; j++)
  A[i][j] = B[j+1][i];
```

```
Unroll j
for (i=0; i<4; i++)
 for (j=0; j<8; j+=2)
  A[i][j] = B[j+1][i];
  A[i][j+1] = B[j+2][i];
```

```
Unroll-and-jam i
for (i= 0; i<4; i+=2)
 for (j=0; j<8; j++)
   A[i][j] = B[j+1][i];
   A[i+1][j] = B[j+1][i+1];
```


How does Unroll-and-Jam benefit locality?

```
Original:
for (i=0; i<4; i++)
 for (j=0; j<8; j++)
  A[i][j] = B[j+1][i] + B[j+1][i+1];
```

```
Unroll-and-jam i and j loops
for (i=0; i<4; i+=2)
 for (j=0; j<8; j+=2) {
  A[i][j]   = B[j+1][i] + B[j+1][i+1];
  A[i+1][j] = B[j+1][i+1] + B[j+1][i+2];
  A[i][j+1] = B[j+2][i] + B[j+2][i+1];
  A[i+1][j+1] = B[j+2][i+1] + B[j+2][i+2];
}
```

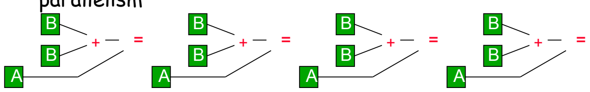
- Temporal reuse of B in registers
- More if I loop is unrolled further

Other advantages of Unroll-and-Jam

```
Original:
for (i=0; i<4; i++)
 for (j=0; j<8; j++)
  A[i][j] = B[j+1][i] + B[j+1][i+1];
```

```
Unroll-and-jam i and j loops
for (i=0; i<4; i+=2)
 for (j=0; j<8; j+=2) {
  A[i][j]   = B[j+1][i] + B[j+1][i+1];
  A[i+1][j] = B[j+1][i+1] + B[j+1][i+2];
  A[i][j+1] = B[j+2][i] + B[j+2][i+1];
  A[i+1][j+1] B[j+2][i+1] + B[j+2][i+2];
}
```

- Less loop control
- Independent computations for instruction-level parallelism



How to determine safety of reordering transformations

- Informally
  - Must preserve relative order of dependence source and sink
  - So, cannot reverse order
- Formally
  - Tracking dependences
  - A simple abstraction: Distance vectors

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?

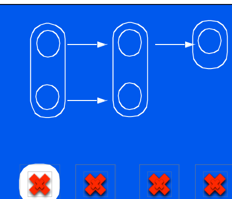
Each iteration can execute independently of others
Free to schedule iterations in any order (e.g., pragma omp forall)

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;
```

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



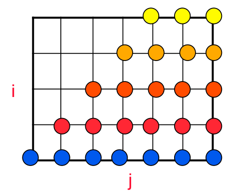
Loop-Carried dependence

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++)
    ...
```

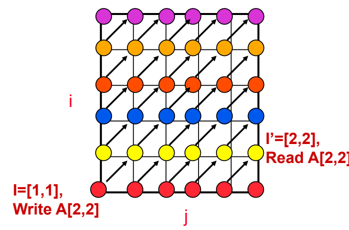


$1 \le i \le 5$
$i \le j \le 7$

- **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 ($i_1, \ldots, i_{c-1}$) = ($i'_1, \ldots, i'_{c-1}$) 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;
```



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)

$$\begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}0\\2\end{bmatrix} \quad \begin{bmatrix}0\\n\end{bmatrix} \quad \begin{bmatrix}1\\-n\end{bmatrix} \quad \begin{bmatrix}1\\0\end{bmatrix} \quad \begin{bmatrix}1\\n\end{bmatrix} \quad \begin{bmatrix}n\\-n\end{bmatrix} \quad \begin{bmatrix}n\\0\end{bmatrix} \quad \begin{bmatrix}n\\n\end{bmatrix}$$

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

0	=
+	<
-	>
+/-	*

Parallelization Test: 1-Dimensional Loop

- **Examples:**

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

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

- **Dependence (Distance) Vectors?**
- **Test for parallelization:**
  - A 1-d loop is parallelizable if for all data dependences $D \in \mathbf{D}$, $D = 0$

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 vectors?**
- **Definition:**
  $D = (d_1, \dots d_n)$ 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 $D = (d_1, \dots d_n)$,
either
$(d_1, \dots d_{i-1}) > 0$
or
$(d_1, \dots d_i) = 0$

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

  $(d_1, \dots d_{i-1}) > 0$

  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=1; j<6; j++)
    A[i+1][j-1]=A[i][j]+B[j];
```

- Distance vectors

- Ok to permute?

Safety of Tiling

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

  OR
  - strip size less than dependence distance

Safety of Unroll-and-Jam

- Unroll-and-jam = tile + unroll
  - Permutation must be legal

  OR
  - unroll less than dependence distance

Unroll-and-jam = tile + unroll?

```
Original:
for (i=0; i<4; i++)
 for (j=0; j<8; j++)
  A[i][j] = B[j+1][i];
```

```
Tile i loop:
for (ii=0; ii<4; ii+=2)
 for (j=0; j<8; j++)
   for (i=ii; i<ii+2; i++)
      A[i][j] = B[j+1][i];
```

```
Unroll i tile:
for (ii= 0; ii<4; ii+=2)
 for (j=0; j<8; j++)
   A[i][j] = B[j+1][i];
   A[i+1][j] = B[j+1][i+1];
```