CS4230 Parallel Programming

Lecture 7: Loop Scheduling cont., and Data Dependences

Mary Hall

September 7, 2012

Administrative

- I will be on travel on Tuesday, September 11
- There will be no lecture
- Instead, you should watch the following two videos:
  - "**Intro to Parallel Programming . Lesson 3, pt. 1 - Implementing Domain Decompositons with OpenMP**", by Clay Breshears, Intel, http://www.youtube.com/watch?v=9n1b9Uwvgaw&feature=relmfu (9:53), reinforces OpenMP data parallelism
  - "Dense Linear Algebra" by Jim Demmel, CS237 at UC Berkeley, http://www.youtube.com/watch?v=2pFOivcJ74g&feature=relmfu (start at 18:15 until 30:40), establishes bounds of dense linear algebra performance, taking parallelism and locality into account
  - There will be a quiz on this material on Thursday, September 13 (participation grade)

Homework 2: Mapping to Architecture

Due before class, Thursday, September 6

Objective: Begin thinking about architecture mapping issues

Turn in electronically on the CADE machines using the handin program: "handin cs4230 hw2 <probfile>"

- Problem 1: (2.3 in text) [Locality]
- Problem 2: (2.8 in text) [Caches and multithreading]
- Problem 3: [Amdahl's Law] A multiprocessor consists of 100 processors, each capable of a peak execution rate of 20 Gflops. What is performance of the system as measured in Gflops when 20% of the code is sequential and 80% is parallelizable?
- Problem 4: (2.16 in text) [Parallelization scaling]
- Problem 5: [Buses and crossbars] Suppose you have a computation that uses two vector inputs to compute a vector output, where each vector is stored in consecutive memory locations. Each input and output location is unique, but data is loaded/stored from cache in 4-word transfers. Suppose you have P processors and N data elements, and execution time is a function of time L for a load from memory and time C for the computation. Compare parallel execution time for a shared memory architecture with a bus (Nehalem) versus a full crossbar (Niagara) from Lecture 3, assuming a write back cache that is larger than the data footprint.

Programming Assignment 1: Due Friday, Sept. 14, 11PM MDT

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

1. Write a program to calculate π 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. You can do this by either decreasing the number of threads or increasing the number of iterations.
- Report results for two different # of threads, holding iterations fixed, and two different # of iterations holding threads fixed. Also report lines of code for the two solutions.

Extra credit: Rewrite both codes using a cyclic distribution and measure performance for same configurations.

Programming Assignment 1, cont.

- A test harness is provided in pi-test-harness.c that provides a sequential pi calculation, 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 pi-openmp –O3 –xopenmp pi-openmp.c
  - Compile w/ Pthreads:
    cc –o pi-pthreads –O3 pi-pthreads.c –lpthread
  - Run OpenMP version: ./pi-openmp > openmp.out
  - Run Pthreads version: ./pi-pthreads > pthreads.out
- Note that editing on water is somewhat primitive – I'm using vim. You may want to edit on a different CADE machine.

Estimating π

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

```
double factor = 1.0;
double sum = 0.0;
for (k = 0; k < n; k++) {
   sum += factor/(2*k+1);
   factor = -factor;
}
pi_approx = 4.0*sum;
```


Today's Lecture

- OpenMP parallel, parallel for and for constructs from last time
- OpenMP loop scheduling demonstration
- Data Dependences
  - How compilers reason about them
  - Formal definition of reordering transformations that preserve program meaning
  - Informal determination of parallelization safety
- Sources for this lecture:
  - Notes on website

OpenMP Sum Examples: Improvement

Last time (sum v3):

```
int sum, mysum[64];
sum = 0;
#pragma omp parallel
{
  int my_id = omp_get_thread_num();
  mysum[my_id] = 0;
}
#pragma omp parallel for
for (i=0; i<size; i++) {
   int my_id = omp_get_thread_num();
   mysum[my_id] += _iplist[i];
}
#pragma omp parallel {
  int my_id = omp_get_thread_num();
  #pragma omp critical
  sum += mysum[my_id];
}
return sum;
```

Improved, posted (sum v3):

```
int sum, mysum[64];
sum = 0;
#pragma omp parallel
{
  int my_id = omp_get_thread_num();
  mysum[my_id] = 0;
  #pragma omp for
  for (i=0; i<size; i++) {
    mysum[my_id] += _iplist[i];
  }
  #pragma omp critical
  sum += mysum[my_id];
}
return sum;
```

Common Data Distributions

- Consider a 1-Dimensional array to solve the global sum problem, 16 elements, 4 threads

CYCLIC (chunk = 1) (version 2):

```
for (i = 0; i<blocksize; i++)
    … in [i*blocksize + tid];
```

BLOCK (chunk = 4) (default, version 1):

```
for (i=tid*blocksize; i<(tid+1) *blocksize; i++)
    … in[i];
```

BLOCK-CYCLIC (chunk = 2) (version 3):

The Schedule Clause

- Default schedule:

```
      sum = 0.0;
#     pragma omp parallel for num_threads(thread_count) \
         reduction(+:sum)
      for (i = 0; i <= n; i++)
         sum += f(i);
```

```
      sum = 0.0;
#     pragma omp parallel for num_threads(thread_count) \
         reduction(+:sum) schedule(static,1)
      for (i = 0; i <= n; i++)
         sum += f(i);
```


OpenMP environment variables

OMP_NUM_THREADS

- sets the number of threads to use during execution
- when dynamic adjustment of the number of threads is enabled, the value of this environment variable is the maximum number of threads to use
- For example,

```
setenv OMP_NUM_THREADS 16 [csh, tcsh]
export OMP_NUM_THREADS=16 [sh, ksh, bash]
```

OMP_SCHEDULE (version 4)

- applies only to `do/for` and `parallel do/for` directives that have the schedule type `RUNTIME`
- sets schedule type and chunk size for all such loops
- For example,

```
setenv OMP_SCHEDULE GUIDED,4 [csh, tcsh]
export OMP_SCHEDULE= GUIDED,4 [sh, ksh, bash]
```


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.

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?

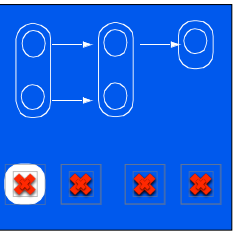
Each iteration can execute independently of others
Free to schedule iterations in any order (e.g., pragma omp forall)

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