CS4961 Parallel Programming

Lecture 14: Reasoning about Performance

Mary Hall
October 7, 2010

Administrative: What's Coming

- Programming assignment 2 due Friday, 11:59PM
- Homework assignment out on Tuesday, Oct. 19 and due Monday, October 25
- Midterm Quiz on October 26
- Start CUDA after break
- Start thinking about independent/group projects

Today's Lecture

- Estimating Locality Benefits
- Finish OpenMP example
- Ch. 3, Reasoning About Performance

Programming Assignment 2: Due 11:59 PM, Friday October 8

Combining Locality, Thread and SIMD Parallelism:

The following code excerpt is representative of a common signal processing technique called convolution. Convolution combines two signals to form a third signal. In this example, we slide a small (32x32) signal around in a larger (4128x4128) signal to look for regions where the two signals have the most overlap.

```
for (l=0; l<N; l++) {
  for (k=0; k<N; k++) {
    C[k][l] = 0.0;
    for (j=0; j<W; j++) {
      for (i=0; i<W; i++) {
        C[k][l] += A[k+i][l+j]*B[i][j];
      }
    }
  }
}
```

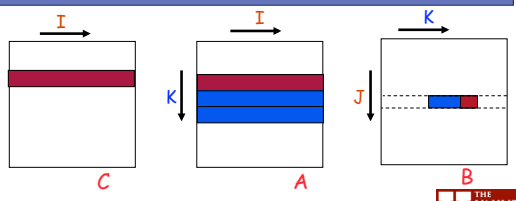
How to use Tiling

- What data you are trying to get to reside in cache?
- Is your tiling improving whether that data is reused out of cache?
- Do you need to tile different loops? Do you need to tile multiple loops? Do you need a different loop order after tiling?
- How big are W and N?
- The right way to approach this is to reason about what data is accessed in the innermost 2 or 3 loops after tiling. You can permute to get the order you want.
  - Let's review Matrix Multiply from Lecture 12
  - Calculate "footprint" of data in innermost loops not controlled by N, which is presumably very large
  - Advanced idea: take spatial locality into account

Locality + SIMD (SSE-3) Example

Example: matrix multiply

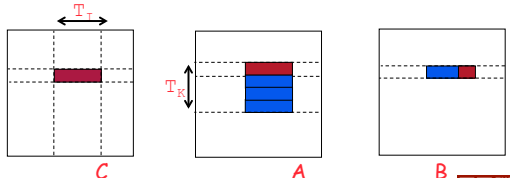
```
for (J=0; J<N; J++)
  for (K=0; K<N; K++)
    for (I=0; I<N; I++)
      C[J][I]= C[J][I] + A[K][I] * B[J][K]
```



Locality + SIMD (SSE-3) Example

Tiling inner loops I and K (+permutation)

```
for (K = 0; K<N; K+=T_K)
  for (I = 0; I<N; I+=T_I)
    for (J =0; J<N; J++)
      for (KK = K; KK<min(K+T_K, N); KK++)
        for (II = I; II<min(I+ T_I, N); II++)
          C[J][II] = C[J][II] + A[KK][II] * B[J][KK];
```



Motivating Example: Linked List Traversal

```
.....
while(my_pointer) {
    (void) do_independent_work (my_pointer);
    my_pointer = my_pointer->next ;
} // End of while loop
.....
```

- How to express with parallel for?
  - Must have fixed number of iterations
  - Loop-invariant loop condition and no early exits
- Convert to parallel for
  - A priori count number of iterations (if possible)

OpenMP 3.0: Tasks!

```
my_pointer = listhead;
#pragma omp parallel   {
  #pragma omp single nowait      {
     while(my_pointer) {
       #pragma omp task firstprivate(my_pointer)          {
         (void) do_independent_work (my_pointer);
        }
        my_pointer = my_pointer->next ;
     }
  } // End of single - no implied barrier (nowait)
} // End of parallel region - implied barrier here
```

firstprivate = private and copy initial value from global variable
lastprivate = private and copy back final value to global variable

Chapter 3: Reasoning about Performance

- Recall introductory lecture:
  - Easy to write a parallel program that is slower than sequential!
- Naïvely, many people think that applying P processors to a T time computation will result in T/P time performance
- Generally wrong
  - For a few problems (Monte Carlo) it is possible to apply more processors directly to the solution
  - For most problems, using P processors requires a paradigm shift, additional code, "communication" and therefore overhead
  - Also, differences in hardware
  - Assume "P processors => T/P time" to be the best case possible
  - In some cases, can actually do better (why?)

Sources of Performance Loss

- Overhead not present in sequential computation
- Non-parallelizable computation
- Idle processors, typically due to load imbalance
- Contention for shared resources

Sources of parallel overhead

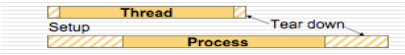
- Thread/process management (next few slides)
- Extra computation
  - Which part of the computation do I perform?
  - Select which part of the data to operate upon
  - Local computation that is later accumulated with a reduction
  - …
- Extra storage
  - Auxiliary data structures
  - "Ghost cells"
- "Communication"
  - Explicit message passing of data
  - Access to remote shared global data (in shared memory)
  - Cache flushes and coherence protocols (in shared memory)
  - Synchronization (book separates synchronization from communication)

Processes and Threads (& Filaments…)

- Let's formalize some things we have discussed before
- Threads …
  - consist of program code, a program counter, call stack, and a small amount of thread-specific data
  - share access to memory (and the file system) with other threads
  - communicate through the shared memory
- Processes …
  - Execute in their own private address space
  - Do not communicate through shared memory, but need another mechanism like message passing; shared address space another possibility
  - Logically subsume threads
  - Key issue: How is the problem divided among the processes, which includes data and work

Comparison

- Both have code, PC, call stack, local data
  - Threads -- One address space
  - Processes -- Separate address spaces
  - Filaments and similar are extremely fine-grain threads
- Weight and Agility
  - Threads: lighter weight, faster to setup, tear down, more dynamic
  - Processes: heavier weight, setup and tear down more time consuming, communication is slower

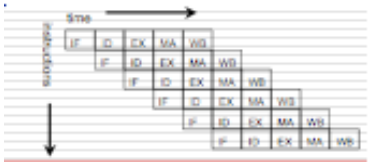


Latency vs. Throughput

- Parallelism can be used either to reduce latency or increase throughput
  - Latency refers to the amount of time it takes to complete a given unit of work (speedup).
  - Throughput refers to the amount of work that can be completed per unit time (pipelining computation).
- There is an upper limit on reducing latency
  - Speed of light, esp. for bit transmissions
  - In networks, switching time (node latency)
  - (Clock rate) x (issue width), for instructions
  - Diminishing returns (overhead) for problem instances
  - Limitations on #processors or size of memory
  - Power/energy constraints

Throughput Improvements

- Throughput improvements are often easier to achieve by adding hardware
  - More wires improve bits/second
  - Use processors to run separate jobs
  - Pipelining is a powerful technique to execute more (serial) operations in unit time
- Common way to improve throughput
  - Multithreading (e.g., Nvidia GPUs and Cray El Dorado)



Latency Hiding from Multithreading

- Reduce wait times by switching to work on different operation
  - Old idea, dating back to Multics
  - In parallel computing it's called *latency hiding*
- Idea most often used to lower λ costs
  - Have many threads ready to go …
  - Execute a thread until it makes nonlocal ref
  - Switch to next thread
  - When nonlocal ref is filled, add to ready list

Interesting phenomenon: Superlinear speedup

Why might Program 1 be exhibiting superlinear speedup?

Different amount of work?

Cache effects?

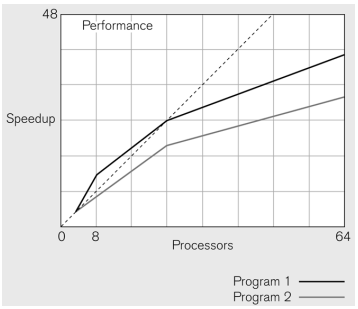


Figure 3.5 from text A typical speedup graph showing performance for two programs; the dashed line represents linear speedup.

Performance Loss: Contention

- Contention -- the action of one processor interferes with another processor's actions -- is an elusive quantity
  - Lock contention: One processor's lock stops other processors from referencing; they must wait
  - Bus contention: Bus wires are in use by one processor's memory reference
  - Network contention: Wires are in use by one packet, blocking other packets
  - Bank contention: Multiple processors try to access different locations on one memory chip simultaneously

Performance Loss: Load Imbalance

- Load imbalance, work not evenly assigned to the processors, underutilizes parallelism
  - The assignment of work, not data, is key
  - Static assignments, being rigid, are more prone to imbalance
  - Because dynamic assignment carries overhead, the quantum of work must be large enough to amortize the overhead
  - With flexible allocations, load balance can be solved late in the design programming cycle

Scalability Consideration: Efficiency

- Efficiency example from textbook, page 82
- Parallel Efficiency = Speedup / NumberofProcessors
  - Tells you how much gain is likely from adding more processors
- Assume for this example that overhead is fixed at 20% of T_S
- What is speedup and efficiency of 2 processors? 10 processors? 100 processors?

Summary:

- Issues in reasoning about performance
- Finish your assignment (try tiling!)
- Have a nice fall break!