Lecture 4

Performance programming and the memory hierarchy;
Performance metrics and measurement;
Cache coherence and consistency
Announcements

• Turnin has been enabled for Programming Lab #1
  ⊕ Within /class/public/cse260fa10/scripts
  ⊕ Instructions: turnin_help.txt
  ⊕ Turnin script: TURNIN_CSE260.A1
• Due on Thursday at 9pm
• Be sure to try the turnin process ahead of time, to be sure you have all required files: teameval.txt, etc.
Revisiting OpenMP
#ifdef _OPENMP
#include <omp.h>

int nthreads = 1;
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    if (tid == 0) {
        nthreads = omp_get_num_threads();
        printf("Number of openMP threads: %d\n", nthreads);
    }
}
#endif
Computational loop

FLOAT c = 1 / 6.0, h = 1.0, c2 = h * h;

for (it= 0; it<nIters; it++) {
#pragma omp parallel shared(U,Un,b,nx,ny,nz,c2,c) private(i,j,k)
#pragma omp for schedule(static,bi)
    for (int i=1; i<=nx; i++)
        for (int j=1; j<=ny; j++)
            for (int k=1; k<=nz+1; k++)
                Un[i][j][k] = c * (U[i-1][j][k] + U[i+1][j][k] + U[i][j-1][k] + U[i][j+1][k] +
                                   U[i][j][k-1] + U[i][j][k+1] - c2*b[i-1][j-1][k-1]);

    Grid3D tmp = U;
    U = Un;
    Un = tmp
}
Computing the residual

FLOAT resid7(Grid3D U, Grid3D B, const int nx, const int ny, const int nz){
double c = 1 / 6.0, err=0;
#pragma omp parallel shared(U,B,c)
#pragma omp for reduction(+:err)
for (int i=1; i<=nx; i++)
    for (int j=1; j<=ny; j++)
        for (int k=1; k<=nz; k++){
            FLOAT du = c * (U[i-1][j][k] + U[i+1][j][k] + U[i][j-1][k] +
                           U[i][j+1][k] + U[i][j][k-1] + U[i][j][k+1] - 6.0*B[i-1][j-1][k-1]);
            FLOAT r = B[i-1][j-1][k-1] - du;
            err = err +  r*r;
        }
return sqrt(err)/(float)((nx+2)*(ny+2)*(nz+2));
}
Caches
Coherency,
Consistency,
False Sharing
The 3 C’s of cache misses

- Cold Start
- Capacity
- Conflict
Nehalem’s memory hierarchy

- **Source:** *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Table 2.7
- All data caches are allocate-on-write

<table>
<thead>
<tr>
<th>Latency (cycles)</th>
<th>Associativity</th>
<th>Line size (bytes)</th>
<th>Write update policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-inclusive</td>
<td>4</td>
<td>8</td>
<td>Writeback</td>
</tr>
<tr>
<td>Non-inclusive</td>
<td>10</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Inclusive</td>
<td>35+</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

4MB for Gainestown

realworldtech.com
Stencil Methods – 2D

\[ \text{Data block} \]

\[ \text{Linear array space} \]
Reducing conflict misses

- Pad the array with unused cells to change the memory access patterns
- Rivera & Tseng [Sigplan, 1998]
- Any other ways?
Cache coherence and consistency
Cache Coherence

- A central design issue in shared memory architectures
- Processors may read and write the same cached memory location
- If one processor writes to the location, all others must eventually see the write

\[
X := 1 \quad \text{Memory}
\]
Cache Coherence

- P1 & P2 load X from main memory into cache
- P1 stores 2 into X
- The memory system doesn’t have a coherent value for X

\[
\begin{align*}
X \leftarrow 1 & \quad \text{Memory} \\
X \leftarrow 2 & \quad \text{P1} \\
X \leftarrow 1 & \quad \text{P2}
\end{align*}
\]
Cache Coherence Protocols

- Ensure that all processors *eventually* see the same value
- Two policies
  - Update-on-write (implies a write-through cache)
  - Invalidate-on-write
SMP architectures

- Employ a *snooping protocol* to ensure coherence
- Processors listen to bus activity
Memory consistency and correctness

- Cache coherence tells us that memory will eventually be consistent
- The memory consistency policy tells us when this will happen
- Even if memory is consistent, changes don’t propagate instantaneously
- These give rise to correctness issues involving program behavior
Memory consistency

- A memory system is consistent if the following 3 conditions hold
  - Program order
  - Definition of a coherent view of memory
  - Serialization of writes
Program order

• If a processor writes and then reads the same location X, and there are no other intervening writes by other processors to X, then the read will always return the value previously written.

\[ X := 2 \]

\[ P \]

\[ X := 2 \]
Definition of a coherent view of memory

- If a processor $P$ reads from location $X$ that was previously written by a processor $Q$, then the read will return the value previously written, if a sufficient amount of time has elapsed between the read and the write.

\[
\begin{align*}
\text{Load } X \\
X:=1 \\
\text{Memory}
\end{align*}
\]
Serialization of writes

- If two processors write to the same location X, then other processors reading X will observe the same sequence of values in the order written.
- If 10 and then 20 is written into X, then no processor can read 20 and then 10.
Memory consistency model

- The memory consistency model determines when a written value will be seen by a reader
- **Sequential Consistency** maintains a linear execution on a parallel architecture that is consistent with the sequential execution of some interleaved arrangement of the separate concurrent instruction streams
- Expensive to implement
- **Relaxed consistency**
  - Enforce consistency only at well defined times
  - Useful in handling false sharing
False sharing

• Consider two processors that write to different locations mapping to different parts of the same cache line
False sharing

- P0 writes a location
- Assuming we have a write-through cache, memory is updated
False sharing

- P1 reads the location written by P0
- P1 then writes a different location in the same block of memory
False sharing

- P1’s write updates main memory
- Snooping protocol invalidate the corresponding block in P0’s cache
False sharing

Successive writes by P0 and P1 cause the processors to uselessly invalidate one another’s cache
Eliminating false sharing

• Cleanly separate locations updated by different processors
  - Manually assign scalars to a pre-allocated region of memory using pointers
  - Spread out the values to coincide with a cache line boundaries

```c
#pragma omp parallel for
for (i=0; i<N; i++)
  sum[i]++;
```
False sharing and conflict misses

• Boundary values, false sharing
• Large memory access strides, conflict misses
• Compare with distributed memory solution

On a single processor

On multiple processors

Cache block
straddles partition
boundary

Contiguity in memory layout

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
& Gupta

Parallel
Computer
Architecture,
Culler, Singh,
Performance metrics
Measures of Performance

- Why do we measure performance?
- Measures of performance
  - Completion time
  - Processor time product
    \[ \text{Completion time} \times \# \text{processors} \]
  - Throughput: amount of work that can be accomplished in a given amount of time
  - Relative performance: given a reference architecture or implementation
    AKA *Speedup*
Parallel Speedup and Efficiency

• How much of an improvement did our parallel algorithm obtain over the serial algorithm?
• Define the *parallel speedup*, $S_P$

$$S_P = \frac{\text{Running time of the best serial program on 1 processor}}{\text{Running time of the parallel program on P processors}}$$

• $T_1$ is defined as the running time of the “best serial algorithm”
• In general: *not* the running time of the parallel algorithm on 1 processor
• **Definition:** *Parallel efficiency* $E_P = S_P/P$
Performance Anomalies

- **Super-linear speedup**: $S_p > P$
- Is it real?
- A better serial algorithm may be lurking
What can go wrong with speedup?

- Not always an accurate way to compare different algorithms….
- … or the same algorithm running on different machines
- We might be able to obtain a better running time even if we lower the speedup
- For an individual user the bottom line is running time $T_p$ or the *space time cost* $P \cdot T_p$
Superlinear speedup

- We have a *super-linear* speedup when

\[ E_P > 1 \implies S_P > P \]

- Super-linear speedups are often an artifact of inappropriate measurement technique
- Where there is a super-linear speedup, a better serial algorithm may be lurking
Scalability

- A computation is **scalable** if performance increases as a “nice function” of the number of processors, e.g. linearly
- In practice scalability can be hard to achieve
  - Serial sections: code that runs on only one processor
  - “Non-productive” work associated with parallel execution, e.g. communication
  - Load imbalance: uneven work assignments over the processors
- Some algorithms present intrinsic barriers to scalability leading to alternatives
  
  ```
  for i=0:n-1  sum = sum + x[i]
  ```
Serial Section

• Limits scalability
• Let \( f = \) the fraction of \( T_1 \) that runs serially
• \( T_1 = f \times T_1 + (1-f) \times T_1 \)
• \( T_P = f \times T_1 + (1-f) \times T_1 / P \)

Thus \( S_P = 1 / [f + (1 - f)/p] \)
• As \( P \to \infty, S_P \to 1/f \)
• This is known as Amdahl’s Law (1967)
Amdahl’s law (1967)

• A serial section limits scalability
• Let $f = \text{fraction of } T_1 \text{ that runs serially}$
• *Amdahl's Law (1967)*: As $P \to \infty$, $S_P \to 1/f$
Weak scaling

• Is Amdahl’s law pessimistic?
• Observation: Amdahl’s law assumes that the workload \((W)\) remains fixed
• But parallel computers are used to tackle more ambitious workloads
• If we increase \(W\) with \(P\) we have weak scaling
  \(f\) often decreases with \(W\)
Computing scaled speedup

• Instead of asking what the speedup is, let’s ask how long a parallel program would run on a single processor
  [J. Gustafson 1992]

• Let $T_P = 1$
• $f' = \text{fraction of serial time spent on the parallel program}$
• $T_1 = f' + (1-f') \times P = S'_P = \text{scaled speedup}$
• Scaled speedup is linear in $P$
Isoefficiency

- Consequence of Gustafson’s observation is that we increase N with P
- Kumar: We can maintain constant efficiency so long as we increase N appropriately
- The isoefficiency function specifies the growth of N in terms of P
- If N is linear in P, we have a scalable computation
- Problem: the amount of memory per core is shrinking
Measuring performance
Challenges to measuring performance

• Reproducibility
  - Transient system operating conditions
  - Differing systems or program configuration

• Measurements are imprecise
  - “Heisenberg uncertainty principle:” measurement technique may affect performance
  - Overheads and inaccuracy

• Explain anomalous behavior, but ignore anomalies that are not significant
Complications

• Cost of measuring a full run is prohibitive
  ♦ Ignore startup code if you plan to run for a much longer time in production

• Transient behavior
  ♦ Repeat your measurements
  ♦ “Warm up” the code before collecting measurements
  ♦ Ignore outliers unless their behavior is important to you
  ♦ Average time, maximum time, minimum time?
Measurement collection

- Report the *best* timings
  - Repeat results ×3 to 5 until at least 2 measures agree to within… 5%, 10%
  - Report the minimum time
- Also report outliers
- A scatter plot or error bar can be useful
Why do we take the minimum time?
Measurement errors are not distributed symmetrically.
Timing collection

• Measures of time
  ► Elapsed, or “wall clock” time
  ► CPU time = system + user time
  ► Overhead, resolution, and quantization effects

• Measurement tools
  ► Can be platform dependent, especially library routines
  ► Unix `time` command does a reasonable job for long-running programs
  ► `gettimeofday()`
Enable others to reproduce your results

- Builds confidence within a community
- Report where you ran, software versions, processor, etc.
  - `uname -a`
    - Linux lilliput 2.6.32-24-server #42-Ubuntu SMP Fri Aug 20 15:38:55 UTC 2010 x86_64 GNU/Linux
  - `gcc --version`
    - gcc (Ubuntu 4.4.3-4ubuntu5) 4.4.3
  - `nvcc --version`
    - nvcc: NVIDIA (R) Cuda compiler driver
      - Copyright (c) 2005-2010 NVIDIA Corporation
      - Built on Wed_Sep__8_17:12:45_PDT_2010
      - Cuda compilation tools, release 3.2, V0.2.1221
  - Access processor configuration information
    - Device # 0 has 30 cores
    - Device # 1 has 4 cores
    - Choosing device 0
    - Device is a GeForce GTX 285, capability: 1.3
    - CUDA Driver version: 2030, runtime version: 2030
Fin