Lecture 4

OpenMP (continued)
Performance metrics and measurement
Announcements

• Trestles accounts are becoming available
• Try logging in over the weekend
• Instead of offering 10% extra credit for running on Trestles, you can get extra credit for implementing tiling as described in the paper I posted the other day on Moodle:
• “An Auto-tuning Framework for Parallel Multicore Stencil Computations,” by S. Kamil et al. IPDPS 2010
Dealing with Data Dependences
Loop carried dependences

- OpenMP will dutifully parallelize a loop when you tell it to, even if doing so “breaks” the correctness of the code.

```cpp
double* u = new u[N];
  u[0] = u[N-1] = 1;
  #pragma omp parallel for num_threads(2)
  for (i=1; i<N-1; i++)
    u[i] = (u[i-1]+ u[i-1])/2;
```

- Sometimes we can restructure an algorithm, as in odd/even sorting.
Parallel Sorting

• Sorting is fundamental algorithm in data processing
  ♦ Given an unordered set of keys \( x_0, x_1, \ldots, x_{N-1} \)
  ♦ Return the keys in sorted order
• The keys may be character strings, floating point numbers, integers, or any object for which the relations >, <, and == hold
• We’ll assume integers here
• Will talk about other algorithms later on
Compare and exchange sorts

• Simplest sort, based on the bubble sort algorithm
• The fundamental operation is compare-exchange
• **Compare-exchange(a[j], a[j+1])**
  - swaps its arguments if they are in decreasing order
  - satisfies the post-condition that \( a[j] \leq a[j+1] \)
  - Returns FALSE if a swap was made

```python
for i = N-1 to 1 by -1 do
    done = TRUE;
    for j = 0 to i-1 do // Compare-exchange(a[j], a[j+1])
        if (a[i] < a[j]) { a[i] ↔ a[j];
            done=FALSE; }
    end do
    if (done) break;
end do
```
Loop carried dependencies

- We cannot parallelize bubble sort owing to the *loop carried dependence* in the inner loop.
- The value of $a[j]$ computed in iteration $j$ depends on the $a[i]$ computed in iterations $0, 1, \ldots, j-1$

```plaintext
for i = N-1 to 1 by -1 do
    done = TRUE;
    for j = 0 to i-1 do
        done = Compare-exchange(a[j], a[j+1])
    end do
    if (done) break;
end do
```
Odd/Even sort

- If we re-order the comparisons we can parallelize the algorithm
  - number the points as even and odd
  - alternate between sorting the odd and even points
- This algorithm parallelizes since there are no loop carried dependences
- All the odd (even) points are decoupled

\[ a_{i-1} \quad a_i \quad a_{i+1} \]
The algorithm

done = false;

for i = 0 to n-1 do

    for j = 0 to n-1 by 2 do // Even
        done &= Compare-exchange(a[j], a[j+1]);
    end do

    for j = 1 to n-1 by 2 do // Odd
        done &= Compare-exchange(a[j], a[j+1]);
    end do
    if (done) break;

end do
Odd/Even Sort Code

```c
int done;
for ( s = 0; s < MaxIter; s++ ) {
    done = Sweep(Keys, 0, n);  /* Odd phase */
    done &= Sweep(Keys, 1, n);  /* Even phase */

    if (done){
        break;
    }
}

return done;
```

Code posted in $PUB/Examples/OpenMP
Inside Sweep

```c
int Sweep(int *Keys, int OE, int n) {
    int done = 1;
    #pragma omp parallel for shared(Keys,n) private(i) reduction(*:done)
    for (int i = OE ; i < n; i+=2) {
        if (Keys[i] > Keys[i+1]){
            Keys[i] ↔ Keys[i+1];
            done *= 0;
        }
    }
    return done;
}
```
OpenMP under the hood

- A program begins life as a single “boss” thread
- When the boss encounters a parallel construct, it creates a team of worker threads
- The lexically enclosed program statements execute in parallel by all team members
- When we reach the end of the scope…
  - The team of threads synchronize and are dissolved; they enter a wait state
  - Only the master thread continues,
- Thread teams can be created and dissolved many times during program execution
  
  www.ncsa.uiuc.edu/UserInfo/Resources/Software/Intel/Compilers/10.0/main_cls/mergedProjects/optaps_cls/whskin_homepage.htm

- A clever compiler can avoid so many thread creations and joins
Memory system behavior
Nehalem’s memory hierarchy

- Source: *Intel 64 and IA-32 Architectures Optimization Reference Manual (HW #1)*

### Table 2-7. Cache Parameters of Intel Core i7 Processors

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>First Level Data</td>
<td>32 KB</td>
<td>8</td>
<td>64</td>
<td>4</td>
<td>1</td>
<td>Writeback</td>
</tr>
<tr>
<td>Instruction</td>
<td>32 KB</td>
<td>4</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Second Level</td>
<td>256KB</td>
<td>8</td>
<td>64</td>
<td>$10^1$</td>
<td>Varies</td>
<td>Writeback</td>
</tr>
<tr>
<td>Third Level (Shared L3)</td>
<td>8MB</td>
<td>16</td>
<td>64</td>
<td>35-40+$^2$</td>
<td>Varies</td>
<td>Writeback</td>
</tr>
</tbody>
</table>

**NOTES:**
1. Software-visible latency will vary depending on access patterns and other factors.
2. Minimal L3 latency is 35 cycles if the frequency ratio between core and uncore is unity.
Memory system behavior

- Off processor values surround each local subproblem
- Non-contiguous data
- Inefficient to access values on certain faces/edges; poor utilization of cache
- False sharing of boundary data
False sharing and conflict misses

- Boundary values, false sharing, internal fragmentation
- 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
Memory access patterns – 2D
A 3D stencil
Memory access patterns – 3D
Stencil Memory access patterns in 3D

Rivera and Tseng
Reducing conflict misses

- Pad the array with unused cells to change the memory access patterns
- Rivera & Tseng [ICS ‘99, SC ‘00]
Performance metrics
Measures of Performance

• Why do we measure performance?
• Measures of performance
  - Completion time
  - Processor time product
    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 \textit{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 \( E_P = S_P / P \)
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 \textit{space time cost} $P T_P$
Superlinear speedup

• We have a \textit{super-linear} speedup when
  \[ S_P > P \implies E_P > 1 \]

• 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:
  \[
  \text{for } i=0:n-1 \text{ sum = sum + x[i]}
  \]
Serial Section

• Limits scalability
• Let $f = \text{the fraction of } T_1 \text{ that runs serially}$
• $T_1 = f \times T_1 + (1-f) \times T_1$
• $T_P = f \times T_1 + (1-f) \times T_1 / P$
  Thus $S_P = 1/[f + (1 - f)/p]$
• As $P \rightarrow \infty$, $S_P \rightarrow 1/f$
• 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
  \[ f \text{ often decreases with } W \]
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 $T_P = 1$

• $f' = \text{fraction of serial time spent on the parallel program}$

• $T_I = f' + (1-f') \times P = S'_P = \text{scaled speedup}$

• 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 2.6.35-30-server #61-Ubuntu SMP Tue Oct 11 18:09:44 UTC 2011 x86_64 GNU/Linux
  - `gcc --version`
    - gcc version 4.4.5 (Ubuntu/Linaro 4.4.4-14ubuntu5)
  - `icpc --version`
    - icpc (ICC) 12.0.2 20110112
  - `nvcc --version`
    - Cuda compilation tools, release 4.0, 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