Programming Assignment 1
Due Wednesday, Sept. 21 at 11:59PM

• Logistics:
  - You’ll use water.eng.utah.edu (a Sun Ultrasparc T2), for which all of you have accounts that match the userid and password of your CADE Linux account.
  - Compile using "cc –O3 –xopenmp p01.c"
  - Write the prefix sum computation from HW1 in OpenMP using the test harness found on the website.
  - What is the parallel speedup of your code as reported by the test harness?
  - 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 adjust this by changing numbers of threads, and frequency of synchronization. You may also want to think about reducing the parallelism overhead, as the solutions we have discussed introduce a lot of overhead.
  - What happens when you try different numbers of threads or different schedules?

• What to turn in:
  - Your source code so we can see your solution
  - A README file that describes at least three variations on the implementation or parameters and the performance impact of those variations.
  - handin "cs4961 p1 <gzipped tarfile>"

• Lab hours:
  - Thursday afternoon and Tuesday afternoon

Red/Blue Computation from text

#10 in text on p. 111:
The Red/Blue computation simulates two interactive flows. An \( n \times n \) board is initialized so cells have one of three colors: red, white, and blue, where white is empty, red moves right, and blue moves down. Colors wrap around on the opposite side when reaching the edge.

In the first half step of an iteration, any red color can move right one cell if the cell to the right is unoccupied (white). On the second half step, any blue color can move down one cell if the cell below it is unoccupied. The case where red vacates a cell (first half) and blue moves into it (second half) is okay.

Viewing the board as overlaid with \( t \times t \) tiles (where \( t \) divides \( n \) evenly), the computation terminates if any tile’s colored squares are more than \( c\% \) one color. Use Perl-L to write a solution to the Red/Blue computation.
Steps for Solution

Step 1. Partition global grid for \( n/t \times n/t \) processors
Step 2. Initialize half iteration (red) data structure
Step 3. Iterate within each \( t \times t \) tile until convergence (guaranteed?)
   - Step 4. Initialize data structure -- Compute new positions of red elts & associated white elts
   - Step 5. Communicate red boundary values
   - Step 6. Compute new positions of blue elts & associated white elts
   - Step 7. Communicate blue boundary values
Step 8. Check locally if DONE

Issues to consider

- Termination test
- In-place update? Or update a copy?
- What if communicating boundary values was REALLY expensive (high value of \( \lambda \))?
  - Does that change how you solve the problem?

Common solution in distributed memory: Ghost cells

Local partition corresponds to just the tile
But, we also care about red elts to the left and blue elts above
Add a boundary, a ghost cell

Review: Predominant Parallel Control Mechanisms

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Instruction,</td>
<td>A single thread of control, same computation applied across &quot;vector&quot; elts</td>
<td>Array notation as in Fortran 90:</td>
</tr>
<tr>
<td>Multiple Instruction,</td>
<td></td>
<td>Parallel loop:</td>
</tr>
<tr>
<td>Multiple Data (MIMD)</td>
<td>Multiple threads of control, processors periodically sync</td>
<td>forall (i=0; i&lt;n; i++)</td>
</tr>
<tr>
<td>Single Program,</td>
<td></td>
<td>Processor-specific code:</td>
</tr>
<tr>
<td>Multiple Data (SPMD)</td>
<td>Multiple threads of control, but each processor executes same code</td>
<td>if ($myid == 0$) {}</td>
</tr>
</tbody>
</table>
SIMD and MIMD Architectures: What’s the Difference?

A typical SIMD architecture (a) and a typical MIMD architecture (b).

Slide source: Grama et al., Introduction to Parallel Computing, http://www.users.cs.umn.edu/~karypis/parbook

Multimedia Extension Architectures

- At the core of multimedia extensions
  - SIMD parallelism
  - Variable-sized data fields:
    - Vector length = register width / type size

Programming Complexity Issues

- High level: Use compiler
  - may not always be successful
- Low level: Use intrinsics or inline assembly tedious and error prone
- Data must be aligned and adjacent in memory
  - Unaligned data may produce incorrect results
  - May need to copy to get adjacency (overhead)
- Control flow introduces complexity and inefficiency
- Exceptions may be masked

1. Independent ALU Ops

\[
R = R + XR \times 1.08327 \\
G = G + XG \times 1.89234 \\
B = B + XB \times 1.29835
\]
2. Adjacent Memory References

\[
\begin{align*}
R &= R + X[i+0] \\
G &= G + X[i+1] \\
B &= B + X[i+2] \\
R &= R + X[i:i+2] \\
G &= G + X[i:i+2] \\
B &= B + X[i:i+2]
\end{align*}
\]

for \(i=0; i<100; i+=1\)

\[
A[i+0] = A[i+0] + B[i+0]
\]

3. Vectorizable Loops

for \(i=0; i<100; i+=4\)

\[
\begin{align*}
A[i+0] &= A[i+0] + B[i+0] \\
\end{align*}
\]

for \(i=0; i<16; i+=4\)

\[
\]

4. Partially Vectorizable Loops

for \(i=0; i<16; i+=1\)

\[
L = A[i+0] - B[i+0] \\
D = D + \text{abs}(L)
\]
4. Partially Vectorizable Loops

for \(i=0; \ i<16; \ i+=2\)
\[
\begin{align*}
L &= A[i+0] - B[i+0] \\
D &= D + \text{abs}(L) \\
L &= A[i+1] - B[i+1] \\
D &= D + \text{abs}(L)
\end{align*}
\]

for \(i=0; \ i<16; \ i+=2\)
\[
\begin{align*}
L_0 &= A[i:i+1] - B[i:i+1] \\
D &= D + \text{abs}(L_0) \\
D &= D + \text{abs}(L_1)
\end{align*}
\]

Exploiting SLP with SIMD Execution

- **Benefit:**
  - Multiple ALU ops → One SIMD op
  - Multiple ld/st ops → One wide mem op

- **Cost:**
  - Packing and unpacking
  - Reshuffling within a register
  - Alignment overhead

Packing/Unpacking Costs

- Packing source operands
  - Copying into contiguous memory

\[
\begin{align*}
C &= A + 2 \\
D &= B + 3
\end{align*}
\]

\[
\begin{align*}
A &= f() \\
B &= g() \\
C &= A + 2 \\
D &= B + 3
\end{align*}
\]
Packing/Unpacking Costs

- Packing source operands
  - Copying into contiguous memory
- Unpacking destination operands
  - Copying back to location

\[
\begin{align*}
A &= f() \\
B &= g() \\
C &= A + 2 \\
D &= B + 3 \\
E &= C / 5 \\
F &= D * 7
\end{align*}
\]

Alignment Code Generation

- Aligned memory access
  - The address is always a multiple of 16 bytes
  - Just one superword load or store instruction

\[
\text{float } a[64]; \\
f \text{or } (i=0; i<64; i+=4) \\
V_a = a[i:i+3];
\]

\[
\begin{array}{cccc}
0 & 16 & 32 & 48 \\
\end{array}
\]

Alignment Code Generation (cont.)

- Misaligned memory access
  - The address is always a non-zero constant offset away from the 16 byte boundaries.
  - Static alignment: For a misaligned load, issue two adjacent aligned loads followed by a merge.

\[
\text{float } a[64]; \\
f \text{or } (i=0; i<60; i+=4) \\
V_a = a[i:i+3];
\]

\[
\begin{array}{cccc}
0 & 16 & 32 & 48 \\
\end{array}
\]

- Statically align loop iterations

\[
\text{float } a[64]; \\
\text{for } (i=0; i<60; i+=4) \\
V_a = a[i+2:i+5];
\]

\[
\text{float } a[64]; \\
\text{for } (i=2; i<62; i+=4) \\
V_a = a[i+2:i+5];
\]
**Alignment Code Generation (cont.)**

- **Unaligned memory access**
  - The offset from 16 byte boundaries is varying or not enough information is available.
  - Dynamic alignment: The merging point is computed during run time.

```plaintext
float a[64];
for (i=0; i<60; i++)
  Va = a[i:i+3];
```

```plaintext
float a[64];
for (i=0; i<60; i++)
  V1 = a[i:i+3];
  V2 = a[i+4:i+7];
  align = (a[i:i+3])%16;
  Va = merge(V1, V2, align);
```

---

**SIMD in the Presence of Control Flow**

```plaintext
for (i=0; i<16; i++)
  if (a[i] != 0)
    b[i]++;
```

```plaintext
for (i=0; i<16; i+=4)
  pred = a[i:i+3] != (0, 0, 0, 0);
  old = b[i:i+3];
  new = old + (1, 1, 1, 1);
  b[i:i+3] = SELECT(old, new, pred);
L1:
```

Overhead:
Both control flow paths are always executed!

---

**An Optimization:**
**Branch-On-Superword-Condition-Code**

```plaintext
for (i=0; i<16; i+=4)
  pred = a[i:i+3] != (0, 0, 0, 0);
  branch-on-none(pred) L1;
  old = b[i:i+3];
  new = old + (1, 1, 1, 1);
  b[i:i+3] = SELECT(old, new, pred);
L1:
```

---

**Control Flow**

- Not likely to be supported in today’s commercial compilers
- Increases complexity of compiler
- Potential for slowdown
- Performance is dependent on input data

Many are of the opinion that SIMD is not a good programming model when there is control flow.
But speedups are possible!
**Nuts and Bolts**

- What does a piece of code really look like?

```cpp
for (i=0; i<100; i+=4)
```

```cpp
for (i=0; i<100; i+=4) {
    __m128 btmp = _mm_load_ps(float B[i]);
    __m128 ctmp = _mm_load_ps(float C[i]);
    __m128 atmp = _mm_add_ps(__m128 btmp, __m128 ctmp);
    void_mm_store_ps(float A[i], __m128 atmp);
}
```

**Wouldn’t you rather use a compiler?**

- Intel compiler is pretty good
  - icc -msse3 -vecreport3 <file.c>
- Get feedback on why loops were not “vectorized”
- First programming assignment
  - Use compiler and rewrite code examples to improve vectorization
  - One example: write in low-level intrinsics