Lecture 10

Image Segmentation
OpenMP
Announcements

- Quiz return
- Performance of A2
- A3 has been posted
Quiz #3

• Of the following types, which are atomic? Assume that all types are non-volatile:
  \texttt{int, long, byte, double, Integer}

• Why does it take longer to increment a variable of type \texttt{SharedLong} (a PJ class) than a variable of type \texttt{long}?

• What guarantee is made by a \textit{happens-before} relationship? A guarantee that memory writes by one specific statement are visible to another specific statement.
## A2 Implementations

<table>
<thead>
<tr>
<th>Original A2</th>
<th>as the software was given to us {{Row}}[Column][Color Plane] where each cell is a short</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D Array of Int</td>
<td>same data structure as Original A2 but with parallel processing implemented</td>
</tr>
<tr>
<td>3D Array of Short</td>
<td>[Row][Column][Color Plane] where each cell is a short</td>
</tr>
<tr>
<td>2D Array of Packed Int</td>
<td>[Row][Column] where each cell is an Int. The Int is subdivided into four sections. First byte is where Red plane value is stored. Second byte is where the Green is stored and third byte is where the Blue is stored. Fourth Byte is not used (wasted).</td>
</tr>
<tr>
<td>1D Array of Short</td>
<td>[Red][Green][Blue] in this array we store pixels by placing each RGB value of a pixel into three cells in the array. We start with row=0 and move to row=height-1 and for each row we start from column=0 and move to column = width − 1.</td>
</tr>
<tr>
<td>1D Array of Packed Int</td>
<td>[Cell] We first pack RGB data into an Int as done in “2D Array of Packed Int” and we store the Int representing the pixel value into 1D array. We start with row=0 and move to row=height-1 and for each row we start from column=0 and move to column = width − 1.</td>
</tr>
<tr>
<td>No Secondary Array (1D Array of Int)</td>
<td>[Red][Green][Blue] Since the Image is displayed using a 1D array of Int, in this case, we don't create a secondary array. We modify and display the same array that is used to update the GUI. In this array we store pixels by placing each RGB value of a pixel into three cells in the array. We start with row=0 and move to row=height-1 and for each row we start from column=0 and move to column = width − 1.</td>
</tr>
</tbody>
</table>
# Performance

<table>
<thead>
<tr>
<th>Min of Time (ms)</th>
<th>Original A2</th>
<th>3D Array of Int</th>
<th>3D Array of Short</th>
<th>2D Array of Packed Int</th>
<th>1D Array of Short</th>
<th>1D Array of Packed Int</th>
<th>1D Array of Int (no Secondary array)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>16549</td>
<td>16403</td>
<td>16538</td>
<td>13009</td>
<td>13840</td>
<td>12377</td>
<td>13552</td>
</tr>
<tr>
<td>p=1</td>
<td>NA</td>
<td>16629</td>
<td>16506</td>
<td>13029</td>
<td>13865</td>
<td>12405</td>
<td>13573</td>
</tr>
<tr>
<td>p=2</td>
<td>NA</td>
<td>9438</td>
<td>9161</td>
<td>6557</td>
<td>6513</td>
<td>6243</td>
<td>6727</td>
</tr>
<tr>
<td>p=4</td>
<td>NA</td>
<td>8000</td>
<td>7991</td>
<td>3203</td>
<td>3348</td>
<td>3177</td>
<td>3831</td>
</tr>
<tr>
<td>p=8</td>
<td>NA</td>
<td>8026</td>
<td>8031</td>
<td>1730</td>
<td>1919</td>
<td>1668</td>
<td>3485</td>
</tr>
<tr>
<td>p=16</td>
<td>NA</td>
<td>8069</td>
<td>8051</td>
<td>1789</td>
<td>2097</td>
<td>1919</td>
<td>3552</td>
</tr>
</tbody>
</table>
Assignment #3

• Image segmentation
Discussion follows
Ilya Kolykhmatov & Vaishali Amin
Image segmentation

Goal: group pixels into regions corresponding to individual surfaces, objects, or natural parts of objects
Connected component labeling

• Simple approach is to solve the connected component labeling problem, using connectivity to join neighboring pixels into connected regions

• All connected pixels get the same label

8-connectivity

A and B are connected
Naïve approach

• Assign unique label to each foreground pixel

What’s the worst case?
Accelerating the labeling process

- Interpret labels as pointers

Assign unique labels

Init pointers

Follow the pointers

Indices

1 2 3 4 5 6 7 8

1 2 3 4 5 6 7 8

before

... after

Follow pointers

Follow pointers

Assign unique labels

Init pointers

Follow the pointers

Indices

1 2 3 4 5 6 7 8

1 2 3 4 5 6 7 8

before

... after

Follow pointers

Follow pointers
Roots

• After local labeling within a block of pixels, all pixels in the same connected component point to their root with the largest label.
Root updates & corrections
Code design
Code Design Roadmap

- Program design
  - Code organization and re-use
  - How to parallelize serial code (code reorganization)
  - How to design with parallelism in mind
  - Parallelism structures / paradigms

- Performance

- Troubleshooting and debugging

- Various choices for multicore processors
  - Explicit threads
  - Higher level models via libraries or compiler/language support
Paradigms

- Client/server
- Divide and Conquer - Data parallel
- Producer/Consumer (Pipelining) - function parallel
Benefits and Pitfalls of Multithreading

• **Benefits**
  – Harness parallelism to improve performance
  – Ability to multitask to realize concurrency, e.g. display

• **Pitfalls**
  – Program complexity
    • Partitioning, synchronization, serial sections, complicated parallel control flow
    • Data dependencies
    • Shared vs. local state (globals like errno)
    • Thread-safe code, re-entrant functions, protect the use of globals
  – New aspects of debugging
    • Race conditions
    • Deadlock

• **OpenMP - a higher level model**
  – Translation from OpenMP to explicit threads
  – Performance
OpenMP programming

• Simpler interface than explicit threads
• Parallelization handled via annotations
  – Loop decomposition
  – Serial sections
  – Barriers
• A standard, language bindings in C/C++, Fortran
  – http://www.openmp.org
• Parallel loop:
  #pragma omp parallel
  {
    #pragma omp for private(i) shared(n)
    for(i=0; i < n; i++)
      work(i);
  }
Work Sharing

- Three different work sharing constructs
  - For
  - Sections
  - Single

https://computing.llnl.gov/tutorials/openMP
Programming model

• Start with a single root thread
• Fan out a set of concurrently executing threads
• Threads may or may not execute on different processors, and might be interleaved
• Scheduling behavior specified separately

```c
#pragma omp parallel // Begin parallel construct
{
    STATEMENTS
}
// End of Parallel Construct; disband the team
```
Parallel Sections

```c
#pragma omp parallel // Begin a parallel construct
{ // form a team
    // Each team member executes the same code
#pragma omp sections // Begin work sharing
{      // A unit of work
    #pragma omp section // A unit of work
    {do something}

    #pragma omp section // Another unit
    {do something else}

} // Wait until both units complete

} // End of Parallel Construct; disband team

// continue serial execution
```
Critical Sections

- Only one thread at a time may run the code in a critical section
- *Mutual exclusion* to implement critical sections

```c
#pragma omp parallel // Begin a parallel construct
{
    #pragma omp sections // Begin worksharing
    { //
        #pragma omp critical // Critical section
        {x = x + 1;}
        #pragma omp critical // Another critical section
        {x = x + 1;}
        ... // More Replicated Code
        #pragma omp barrier // Wait for all members to arrive
        } // Wait until both units of work complete
}
```
Parallel for loop

```c
#pragma omp parallel private(i) shared(n)
{
    #pragma omp for
    for(i=0; i < n; i++)
        work(i);
}
```

- Static workload assignment
- Dynamic assignment for irregular problems (later on)
- Each thread gets a unique range of indices based on its identity
- The calls to work(i) take the same time to compute
- Shorthand for

```c
#pragma omp parallel private(i) shared(n) for
for(i=0; i < n; i++)
    work(i);
```
Image smoothing in OpenMP
Image smoothing

- The update formula

\[
\text{for } t = 1 \text{ to } T \\
\quad \text{for } (i,j) \text{ in } 1:N-2 \times 1:N-2 \\
\quad\quad u^{\text{new}}[i,j] = (u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1]) / 4 \\
\quad u = u^{\text{new}}
\]

\[
\begin{bmatrix}
0 & 1 & 0 \\
1 & -4 & 1 \\
0 & 1 & 0
\end{bmatrix}
\]
Workload decomposition in OpenMP

• Translator automatically generates appropriate local loop bounds
• Here we parallelize the outer loop index

```c
for (t=1; t<= T; t++){
    #pragma omp parallel private(i) shared(n)
    #pragma omp for
    for(i=1; i < n-1; i++)
        for(j=1; j < n-1; j++) {
            \[ u_{new}[i,j] = \frac{(u[i-1,j] + u[i+1,j]+ u[i,j-1]+ u[i, j+1])}{4} \]
        }
}
```

• Generated code

```c
mymin = 1 + ($TID * n/nthreads),     mymax = mymin + n/nthreads -1
for(i=mymin; i < mymax; i++)
    for(j=1; j < n-1; j++) {
        \[ u_{new}[i,j] = \frac{(u[i-1,j] + u[i+1,j]+ u[i,j-1]+ u[i, j+1])}{4} \]
    }
```

0 1 2 3

0
1
2
3
How does it work under the hood?

- When the master thread encounters a parallel construct, it creates a team of threads.
- The enclosed program statements enclosed executed in parallel by all team members, including procedure calls.
- Statements enclosed lexically within a construct define the static extent of the construct.
- When we reach the end of the scope the team of threads synchronize, the team is dissolved, and only the master thread continues execution. The other threads in the team enter a wait state.
- Thread teams can be created and dissolved many times during program execution.

[Link](http://www.ncsa.illinois.edu/UserInfo/Resources/Software/Intel/Compilers/10.0/main_cls/mergedProjects/optaps_cls/common/optaps_par_proc.htm)
Generated code

```c
int a, b;
main() {
    // serial segment
    #pragma omp parallel num_threads (8) private (a) shared (b)
    {
        // parallel segment
    }
    // rest of serial segment
}
```

Sample OpenMP program

```c
int a, b;
main() {
    // serial segment
    for (i = 0; i < 8; i++)
        pthread_create (........., internal_thread_fn_name, ...);
    for (i = 0; i < 8; i++)
        pthread_join (.........);
    // rest of serial segment
}

void *internal_thread_fn_name (void *packaged_argument) {
    int a;
    // parallel segment
}
```

Corresponding Pthreads translation

www-users.cs.umn.edu/~karypis/parbook

*Introduction to Parallel Computing*, Grama et al., Addison Wesley, 2003
Partitioning

- Each thread gets \( \frac{N}{NT} \) elements
- What if \( N \% NT \approx 0 \)?
- Let \( q = N \% NT \)
- First \( q \) threads get \( \lceil \frac{N}{NT} \rceil \) elements, others get \( \lfloor \frac{N}{NT} \rfloor \) elements
Nested parallelization

Setenv OMP_NESTED 1

#pragma omp parallel for shared(u, u\text{new}, n) private(i,j)
  schedule(static)
  for(i=1; i<n-1; i++) {
    for(j=1; j<n-1; j++) {
      u\text{new}[i,j] = (u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1]) / 4
    }
  }
}
Reductions in OpenMP

- OpenMP uses a local accumulator, which it then accumulates globally when the loop is over

```c
#pragma omp parallel reduction(+:sum)
for (int i=i0; i< i1; i++)
    sum += x[i];
```

```c
#pragma omp parallel default(private) shared (npoints) reduction(+: sum){
    sum = 0;
    #pragma omp for
    for (i = 0; i < npoints; i++) {
        rand_x = (double)(rand_r(&seed))/(double)((2<<14)-1);
        rand_y = (double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_x - 0.5) * (rand_x - 0.5) +
             (rand_y - 0.5) * (rand_y - 0.5)) < 0.25)
            sum ++;
    }
}
```

www-users.cs.umn.edu/~karypis/parbook

*Introduction to Parallel Computing*, Grama et al., Addison Wesley, 2003
Dynamic scheduling

#pragma omp parallel private(i,j) shared(unew, u, n)
#pragma omp for schedule(dynamic)
for(i=0; i < n; i++)
    for(j=0; j < n; j++) {
        u\text{new}[i,j] = \text{Work}(u[i,j])
    }

setenv OMP_DYNAMMIC 1
An example of false sharing

float a[m,n], s[m]  
// Outer loop is in parallel  
// Consider m=4, 128 byte cache line size  
// Thread i updates element s[i]  
!$omp parallel do private(i,j), shared(s,a)  
  for i = 0, m-1  
    s[i] = 0.0  
    for j = 0, n-1  
      s[i] += a[i,j]  
    end for  
  end for
Removing false sharing

float a[m,n], s[m,32]
!$omp parallel do private(i,j), shared(s,a)
for i = 0, m–1
    s[i,1] = 0.0
for j = 0, n–1
    s[i,1] += a[i,j]
end for
end for

![Graph showing speedup vs processors with three lines: false sharing, after tuning, and ideal speedup.]
Reducing conflict misses

- Pad the array with unused cells to change the memory access patterns
- Rivera & Tseng [Sigplan, 1998]
- Any other ways?
Dependencies can inhibit parallelization

```c
#pragma omp parallel for
for (i=2; i < 10; i++)
    factorial[i] = i * factorial[i-1];
```
What’s wrong with this program?

```c
#pragma omp parallel for
def for (i=0; i < 100; i++) {
    temp = array[i];
    array[i] = do_something(temp);
}
```
Two solutions

#pragma omp parallel for
for (i=0; i < 100; i++) {
    int temp; // variables declared within a parallel construct
    // are, by definition, private
    temp = array[i];
    array[i] = do_something(temp);
}

// This also works. The variable temp is declared private
#pragma omp parallel for private(temp)
for (i=0; i < 100; i++) {
    temp = array[i];
    array[i] = do_something(temp);
}
Removing a dependence

```
#pragma omp parallel for
for (i=0; i < NumElements; i++) {
    array[i] = StartVal;
    StartVal++;
}
```

```
#pragma omp parallel for
for (i=0; i < NumElements; i++){
    array[i] = StartVal + i;
}
StartVal += NumElements;
```
Documentation

• Documentation at software.intel.com/en-us/articles/
  – getting-started-with-openmp
  – more-work-sharing-with-openmp
  – advanced-openmp-programming
  – 32-openmp-traps-for-c-developers

• Tutorial on OpenMP, with code examples

• Additional material
  http://www.cse.ucsd.edu/classes/fa08/cse260/Abe.html#openmp