

CSE 160

Lecture 6

Image smoothing application
Synchronization

Announcements

- We are back to Room 1202 (EBU3B)
 - But not on 3/1, 3/13, 3/15
 - Meet for a double slot on 3/13, 1pm to 3pm (SDSC likely)
 - We add an extra class on a Tuesday or Thursday evening in 1202 to replace 3/1 lecture (TBD)
- Lab hours in the basement of EBU3B
 - New TA-lab sessions 12N-5pm on Tuesday

Today's lecture

- Synchronization under the hood
- Image processing Application

Recall barrier synchronization

- Don't read a location updated by other threads that had not had the chance to produce its contribution (true dependence)
- Don't overwrite the values used by other processes in the current iteration until they have been consumed (anti-dependence)
- No thread can move past a barrier until all have arrived

```
pthread_mutex_lock (&mutex_sum);  
sum += 2*(TID+1);  
pthread_mutex_unlock (&mutex_sum);  
Barrier();  
if (TID == 0)  
    cout << "Total sum is " << sum << endl;
```

Building a linear time barrier with locks

```
Mutex arrival=UNLOCKED, departure=LOCKED; // Shared  
int count=0; // Shared
```

```
void Barrier( )  
    arrival.lock( ); // atomically count the  
    count++; // waiting threads  
    if (count < $NT) arrival.unlock( );  
    else departure.unlock( ); // last processor  
                                // enables all to go  
  
    departure.lock( );  
    count--; // atomically decrement  
    if (count > 0) departure.unlock( );  
    else arrival.unlock( ); // last processor resets state
```

Compare and swap

- We build mutex and other synchronization primitives with special atomic operations, implemented with a single machine instruction, e.g. CMPXCHG
- Do atomically: compare contents of memory location **loc** to **expected**; if they are the same, modify the location with **newval**

```
CAS ( *loc , expected , newval ) {  
    if ( *loc == expected ) {  
        *loc = newval;  
        return 0;  
    }  
    else  
        return 1  
}
```

- We can then build mutexes with CAS

```
Lock( *mutex ) {  
    while (CAS ( *mutex , 1, 0)) ;  
}  
  
Unlock( *mutex ) { *mutex = 1; }
```

Memory fences

- How are we assured that a value updated within a critical section becomes visible to all other threads?
- With a *fence* instruction, e.g. MFENCE
- “A serializing operation guaranteeing that every load and store instruction that precedes, *in program order*, the MFENCE instruction is globally visible before any load or store instruction that follows the MFENCE instruction is globally visible.”
[Intel 64 & IA32 architectures software developer manual]
- Also see www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html

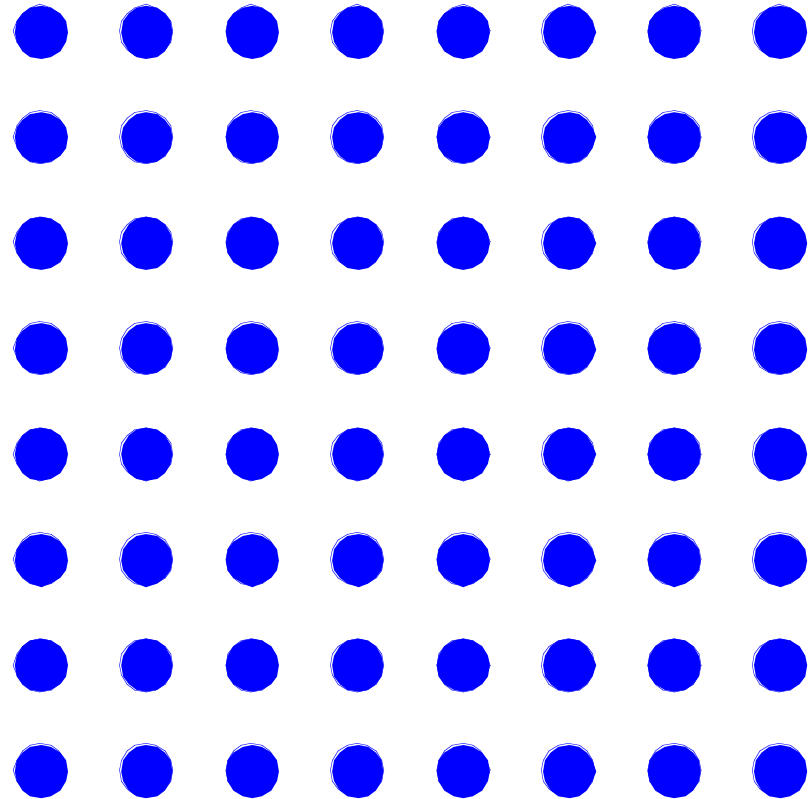
```
pthread_mutex_lock (&mutex);  
sum += local sum;  
pthread_mutex_unlock (&mutex);
```

Today's lecture

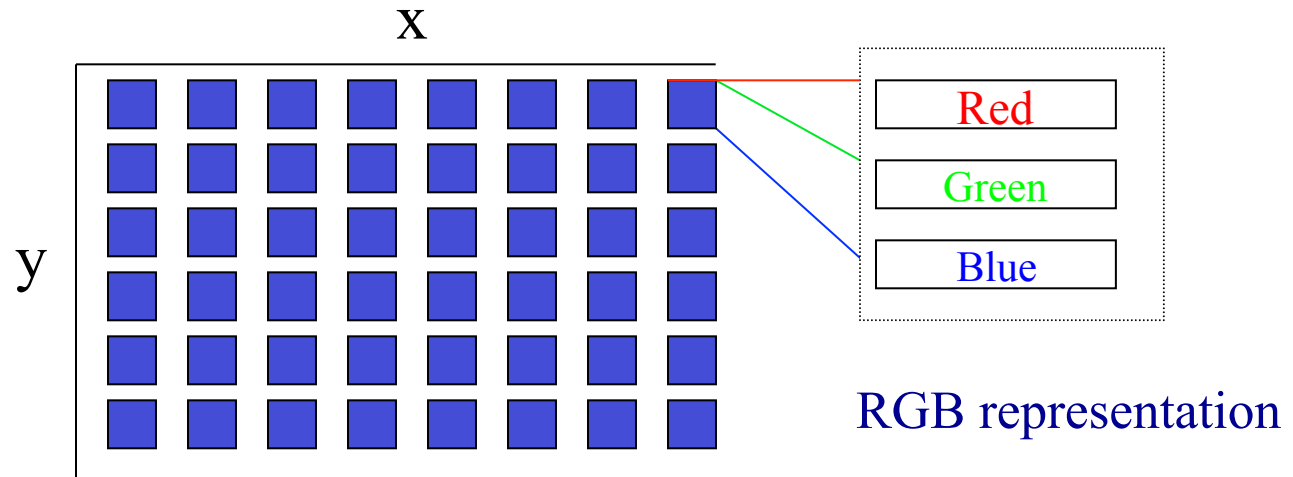
- Synchronization under the hood
- Image processing Application

Stencil methods

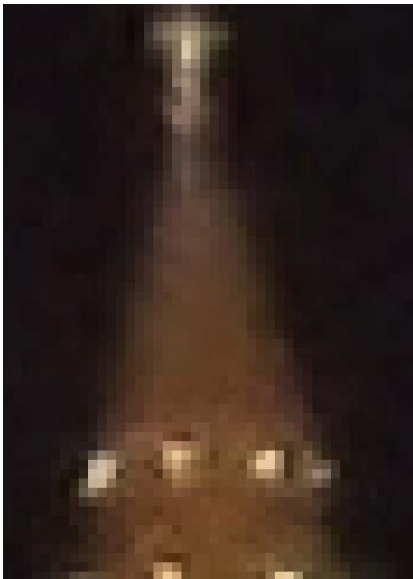
- Many physical problems are simulated on a uniform *mesh* in 1, 2 or 3 dimensions
- *Field variables* defined on a discrete set of points
- A *mapping* from ordered pairs to *physical observables* like temperature and pressure
- Important applications
 - Differential equations
 - Image processing



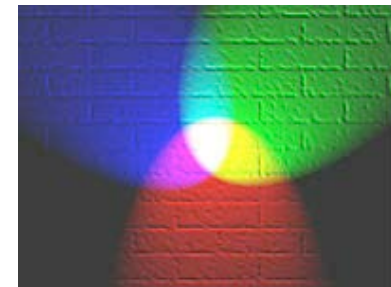
Digital Image Representation



Ryan Cuprak



Photos by Martin Juell



wikipedia

Image smoothing algorithm

- Repeat as many times as necessary

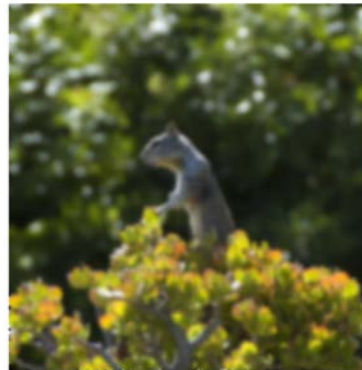
for (i,j) in 0:N-1 x 0:N-1

$$I^{\text{new}}[i,j] = (I[i-1,j] + I[i+1,j] + I[i,j-1] + I[i,j+1])/4$$

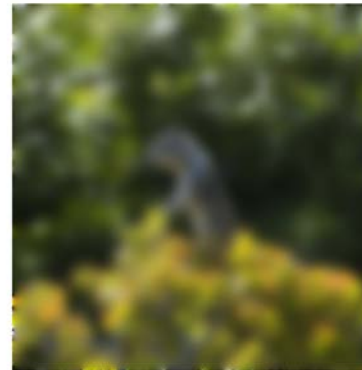
$I = I^{\text{new}}$



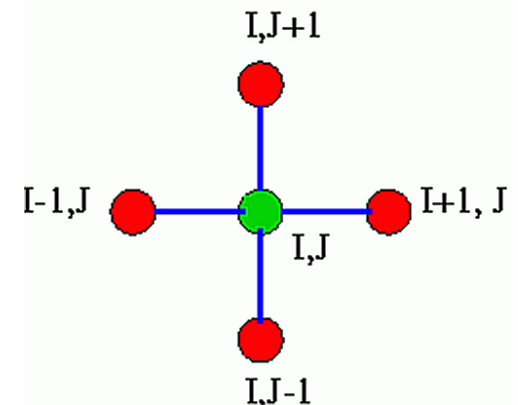
Original



100 iter

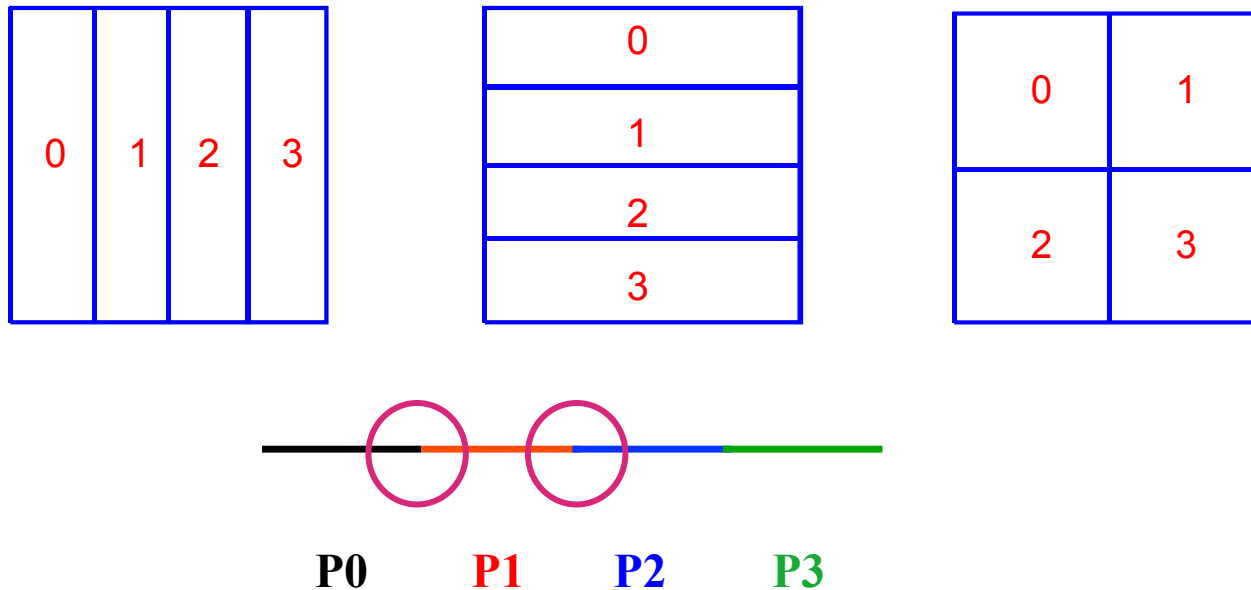


1000 iter



Parallel Implementation

- Partition data into parts, assigning each to a unique thread
- Dependences on values found on neighboring processes
- Threads share boundary values



Multithreaded Smoother()

Global Change, $I[:,:]$, $I^{\text{new}}[:,:]$

Local mymin = 1 + (\$TID * n / \$NT),
mymax = mymin + n / \$NT - 1;

Local done = FALSE;

while (!done) do

Local myChange = 0;

Change = 0;

update I^{new} and myChange

Change += myChange;

if (Change < Tolerance) done = TRUE;

Swap pointers: $I \leftrightarrow I^{\text{new}}$

end while

update I^{new} and myChange:
for i = mymin to mymax do
for j = 1 to n do
 $I^{\text{new}}[i,j] = \dots$
myChange += ($I^{\text{new}} - I^{\text{old}}$)²
end for
end for

0
1
2
3

Is this code correct?

Correctness

Global Change, $I[:,:]$, $I^{\text{new}}[:,:]$

Local $\text{mymin} = 1 + (\$TID * n / \$nprocs)$,

$\text{mymax} = \text{mymin} + n / \$nprocs - 1$;

Local $\text{done} = \text{FALSE}$;

while (!done) do

Local $\text{myChange} = 0$;

BARRIER

Only on thread 0: $\text{Change} = 0$;

BARRIER

update I^{new} and myChange

CRITICAL SEC: $\text{Change} += \text{myChange}$

BARRIER

if ($\text{myChange} < \text{Tolerance}$) $\text{done} = \text{TRUE}$;

Only on thread 0: Swap pointers: $I \leftrightarrow I^{\text{new}}$

end while

0
1
2
3

Does this code use minimal synchronization?