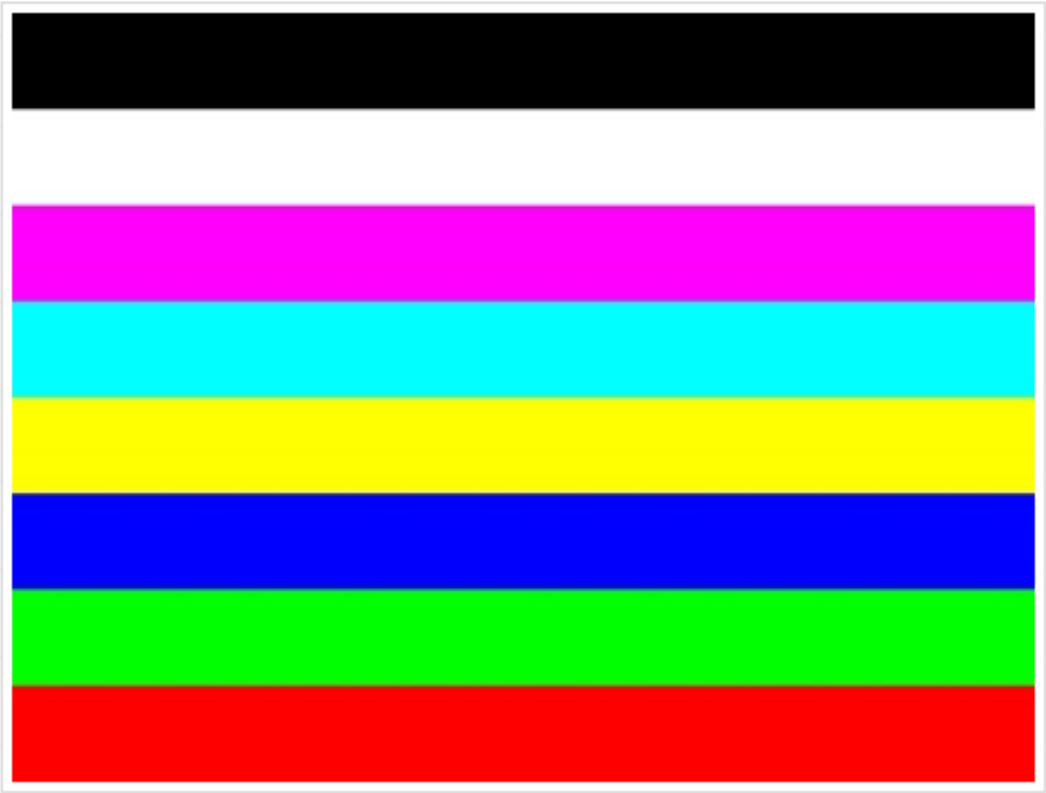
A6. 04

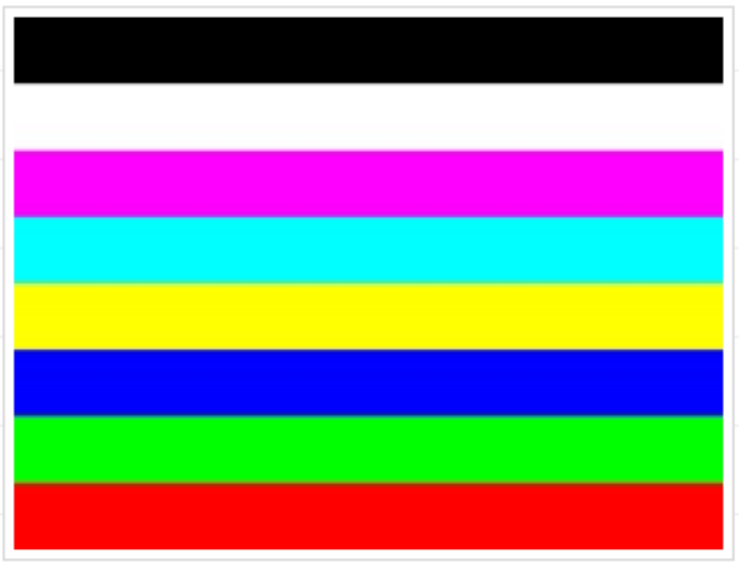
- basic image output program:



- want to split image into horizontal pixel bands so that we can write them independently, IN PARALLEL

- seems straightforward enough:
  - split rows into one band per CPU core,
  - each core writes own row
  - let's say we have ncores
  - then split image into bands: h/ncores

- first obstacle:
  - can't write to file in parallel
  - but can write to memory in parallel, so long as cores don't write to the same pixels

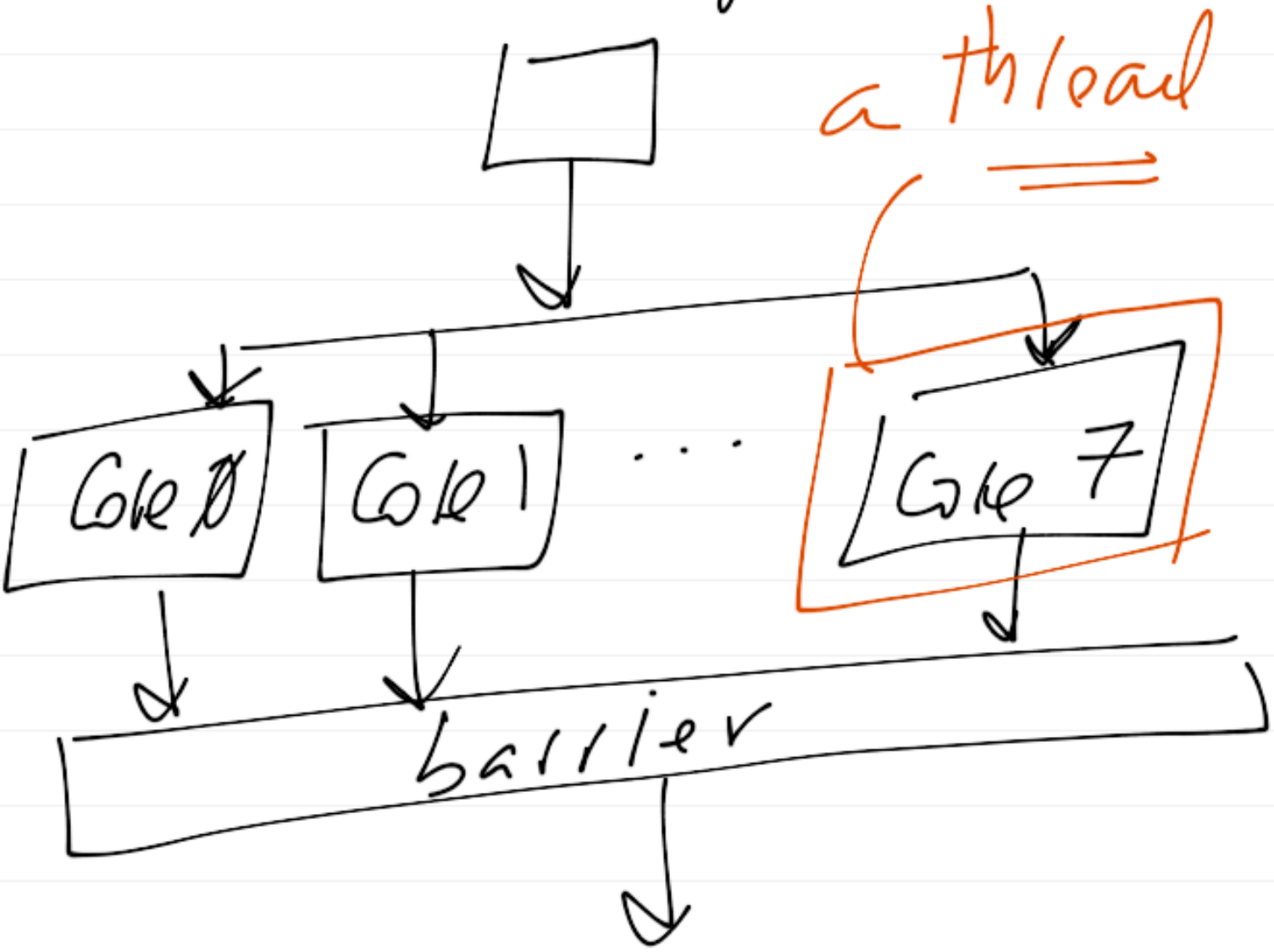


Core 0 → y : 0 to $\frac{h}{n\text{cores}}$

Core 1 → y : $\frac{h}{n\text{cores}}$ + 1 to $\frac{2h}{n\text{cores}}$

and so on ...

- program flow of control:



- not all cores may finish at the same time
- all can access shared memory

- shared memory vs. private memory
- if a core uses index (x, y), it cannot share this info with any other core
- each core gets a private copy of (x, y)

$\Rightarrow$ some variables must be private

- copying memory among cores is expensive
- something like our image (our light_ring array) should be shared — no copies made to cores
- just need to ensure each core writes to non-overlapping memory

- parallelizing code is surprisingly easy, BUT have to be careful about memory access — have to specify what's private and what's shared

- parallelization on multi-core architecture done via OpenMP

- Let's do ASG 04 serially first (all that's required)
- parallel part is optional
- OpenMP won't work with our version of Objective-C (gnustep)
- our gnustep install is not thread-safe

- Serial version of Code:

```
pixel_t *red =
    [(pixel_t *)[pixel_t alloc]
        init: 1.0: 0.0: 0.0];
    :
    :
samp *grn  ( 0, 1, 0)
     *blu  ( 0, 0, 1)
     *ylw  ( 1, 1, 0)
     *cyn  ( 0, 1, 1)
     *mgn  ( 1, 0, 1)
     *wht  ( 1, 1, 1)
     *blk  ( 0, 0, 0)
```

```
pixel_t *pix = ... alloc/init to 0,0,0

int tid, i, x, y;
int w = 640, h = 480;

irgb_t *imgloc, *img;
timer_t *timer =
[(timer_t *)[timer_t alloc]
    init];
```

```
img = (irgb_t *) malloc
      (w*h*sizeof(irgb_t));

[timer start]; // starts timer -
                  just records
                  current time of day
```

```
//main loop
for (y = h-1; y >= 0; y--) {
    //quick hack to simulate 8 cores
    tid = y/(h/8);
    for (x = 0; x < w; x++) {
        [pix autorelease];
        switch(tid) {
            case 0: pix = [red copy];
                break;
            case 1: pix = [grn copy];
                break;
```

```
        :
        case 7: pix=[blk copy];
            break;
    ]//switch

    imgloc= img + y*w + x;
    for(i=0; i<3; i++)
      (*imgloc)[i]=
[[pixscale:255.0] get:i];
    } // Inner for
] //outer for
```

```
[timer stop];
fprintf(stderr, "Cores: 1\n");
fprintf(stderr, %.2lf msec\n",
        [timer elapsed_msec]);
// write out image
fprintf(stdout, "P6 %d %d 255\n",
                w, h);
```

```
for(y=h-1; y>=0; y--) {
  for(x=0; x<w; x++) {
    imgloc = img + y*w + x;
    fwrite(imgloc,
      sizeof(irgb_t), 1, stdout);
  }
}
```

```
// free memory
[red release];
[grn release];
:
[blu release];

free(img);

[pool drain];
}
```

- pretty basic program
- want to parallelize
  first nested loop
  (writing to memory)
  but not the second
  (writing to file)

- first, need:

```
#include <omp.h>
```

- then,

```
int chunk, ncores=1;

#pragma omp parallel private(tid)
{
  tid=omp_get_thread_num();
  if(tid==0)
    ncores=
        omp_get_num_threads();
}
```

```
chunk = h / ncores; }
```
this will split up rows per core

```
    :
[timer start];

#pragma omp parallel \
        shared(chunk, w, h, img) \
  private(tid, i, x, y, pix, imgloc)
{
```

```
tid = omp_get_thread_num();

#pragma omp for \
    schedule(static, chunk)
for(y=h-1; y>=0; y--){
```

// pragma parallelizes just the above for statement — unrolls loop & splits up chunk iterations per core

- rest of code the same except for closing } that closes

#pragma omp parallel

block just above

[timer stop];

- in theory, code runs

N cores times faster

(in practice there is some overhead)

- what is this timer_t ?

```
@interface timer_t : NSObject
{ double ts; // time start
  double te; // time end
  double tt; // time total
}                (end - start)
```

- init timer to 0.0 for each of ts, te, tt
- use this function to timestamp:
- (double) stamp_us

```
{ double s, us, tod;
  struct timeval tp;
  gettimeofday(&tp, NULL);
  s = (double)(tp.tv_sec);
  us = (double)(tp.tv_usec);
```

```
tod = s * 1000000.0 + us;
```

time of day in microsec

```
return (tod);
}
```

- need

```
#include <sys/time.h>
```

for gettimeofday

- rest of the functions are easy:

```
-(void) start
{ts = [self stamp_us];}

-(void) stop
{te = [self stamp_us];
 tt = te - ts;
}
```

```
- (double) elapsed_us
    { return tt; }

- (double) elapsed_ms
    { return tt / 1000.0; }

- (double) elapsed_s
    { return tt / 1000000.0; }
```

- for openMP to work, need to compile with -fopenmp flag

- pretty easy — BUT, for Obj-C to work, each thread should have its own autorelease pool

- our version of gunstep doesn't support this

- However, gnustep does support some fancy NS code
- let's redo the servid image output program using NS objects:

NSSize
NSBitmapImageRep
NSImage
NSData

- We can use these to write out JPEG images instead of our .ppm ones

- to compile, need:

```
CFLAGS = `gnustep-config
        --objc-flags`
```

backward quotes

```
LDLIBS = `gnustep-config
        --gui-libs`
```

- on Darwin (Macs):

LDLIBS =
  - framework Foundation \
  - framework AppKit
- at top of main.m:

```
#import <Foundation/Foundation.h>
#import <AppKit/NSImage.h>
#import <AppKit/NSGraphics.h>
```

```
#ifndef __APPLE__
#ifndef NSUInteger
#define NSUInteger
        unsigned int
#endif
#endif
```

```
int x,y, w=640, h=480;
NSSize size = NSMakeSize(w, h);
NSBitmapImageRep *bitmap =
  [[NSBitmapImageRep alloc]
   initWithBitmapDataPlanes: NULL
   pixelsWide: size.width
   pixelsHigh: size.height
   bitsPerSample: 8
   samplesPerPixel: 3
```

hasAlpha: NO
isPlanar: NO
colorSpaceName:
NSDeviceRGBColorSpace
bitmapFormat: 0
bytesPerRow: size.width * 3
bitsPerPixel: 24];

```
NSUInteger red[3] = {255, 0, 0};
NSUInteger grn[3] = {0, 255, 0};
    .
    .
    .
UIImage *img = [NSImage alloc
 init WithData:
 [bitmap TIFFRepresentation]];
```

- now the for loops:

```
for(y=h-1; y>=0; y--) {
  tid = y/(h/8);
  for(x=0; x<w; x++) {
    switch(tid) {
      case 0:
      [bitmap setPixel: red
                    atX: x
                      y: y];
      break;
    } : //same for 1,2,3,...
```

```
NSData *jpgData =
  [bitmap representationUsingType:
      NSJPEGFileType
    properties: nil];
[jpgData writeToFile:
    @"./image.jpg"
  atomically: NO];
```

```
[img release];
[bitmap release];

[pool drain];

}
```

- how to parallelize ?
- would need to setup pixel data before feeding it to bitmap.

- so...
- you can use the above to make your own .jpg images
- a bit more portable than PPMs (e.g., web pages)