

What is Concurrency?

ICS432 Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Disclaimer

- There is content in the two sets of lecture notes in this module that overlaps with ICS332
- If you took ICS332 last semester, bear with us (or zone out)
- But if you took it a long(er) time ago, experience shows this is useful!

Concurrency

- **Definition:** Execution of multiple “tasks” at the “same” time
- You have mostly written non-concurrent, or **sequential**, programs
 - At any point, you could stop the program and say exactly which execution is being executed, what the calling sequence is, what the runtime stack looks like, etc.
 - And there is a single answer to all the above for any execution of your program
- In a concurrent program, you design the program in terms of **tasks**, where each task as a “life of its own”
 - Each task has a specific job to do
 - Tasks may need to “talk” to each other
 - Tasks can be in different regions of the code or in the same region of the code a the same time
 - Tasks can be short-lived or last the whole program’s execution
- A different way of thinking/programming

A brief history of concurrency (1)

- First machines were used in “single-user mode”
 - I declare: “I am going to use the machine for 2PM till 4PM”
 - I go in the special machine room and sit there for 2 hours
 - I try the punch cards that I have prepared in advance
 - I find bugs
 - I debug
 - etc.
- Extreme lack of productivity
 - During my “thinking time”, this multi-million \$ machine does nothing

A brief history of concurrency (2)

- Batch Processing!

- Instead of reserving the machine for a lapse of time to do all my activities (including debugging), I “submit” requests to a “queue”
 - The queue serves requests in order (possibly with priorities)
 - When my program fails and stops, somebody else gets the machine immediately
- Great but: CPU idle during I/O!
 - And I/O takes foreeeeeever

A brief history of concurrency (3)

- Multi-programming (the 60's)
 - Multiple programs reside in memory at once
 - Made possible due to increased memory size
 - Requires interrupts and memory protection
- Time-sharing (the 70's)
 - Multi-programming but rapid alternation between programs
 - Provides the illusion of programs all running simultaneously on the machine
- This is all in ICS332, and is what we have today
 - Virtual memory, fast context switching, etc.
- Eventually this has led to concurrency in user applications!
 - My application is “logically” multiple concurrent tasks
 - I can now implement it as concurrent tasks and the OS will run them simultaneously!
 - This is a main topic of this course

Concurrent Programs

- A program consists of multiple files/modules/classes/functions

```
Terminal - vim -- 98x69
submit_request()
// Low-level function called by most of the functions below
void submit_request(a_host_t scheduler_host, int id, int nodes,
                  double actual_duration, double requested_duration,
                  int port_started, int port_done, int port_queue_size) {
    job_request_t job_request; // for communication with the scheduler
    a_task_t task;

    job_request = (job_request_t) calloc(1, sizeof(_job_request_t));
    job_request->id = id;
    job_request->nodes = nodes;
    job_request->actual_duration = actual_duration;
    job_request->requested_duration = requested_duration;
    job_request->channel_started = port_started;
    job_request->channel_done = port_done;
    job_request->channel_queue_size = port_queue_size;

    task = HSO_task_create("request", id, (void*) job_request);
    if (HSO_task_put(task, scheduler_host, PORT_JOB_REQUEST) != HSO_OK) {
        xbt_assert(0, "Error while sending to scheduler: is on channel PORT_JOB_REQUEST, HSO_host_get_name(scheduler_host)");
    }

    // submit job()
    // submit the request to scheduler(s)
    void submit_job(pending_job_t pending_job,
                  scheduler_info_t* schedulers, int num_schedulers,
                  scheduler_info_t preferred,
                  across_cluster_algorithm_t across_cluster_algorithm,
                  across_cluster_percentage,
                  intra_cluster_algorithm_t intra_cluster_algorithm,
                  int intra_cluster_percentage,
                  int port_started, int port_done, int port_queue_size) {
        int i;
        int num_target_schedulers = 0;
        scheduler_info_t* target_schedulers = NULL;

        // Find out the list of appropriate schedulers
        find_target_schedulers(schedulers, num_schedulers,
                              pending_job, preferred,
                              across_cluster_algorithm, across_cluster_percentage,
                              &num_target_schedulers, &target_schedulers);
        if (num_target_schedulers == 0) {
            xbt_assert(0, "No target scheduler found for job\n");
        }

        // Send requests to each target scheduler
        for (i = 0; i < num_target_schedulers; i++) {
            submit_job_to_target_scheduler(target_schedulers[i], pending_job,
                                         across_cluster_algorithm, across_cluster_percentage,
                                         intra_cluster_algorithm, intra_cluster_percentage,
                                         port_started, port_done, port_queue_size);
        }
        free(target_schedulers);
    }

    // submit job to target scheduler()
    // with possible multiple requests to each
    void submit_job_to_target_scheduler(scheduler_info_t scheduler, pending_job_t pending_job,
                                      across_cluster_algorithm_t across_cluster_algorithm,
                                      int intra_cluster_percentage,
                                      int port_started, int port_done, int port_queue_size) {

```

```
Terminal - vim -- 101x71
print_queue()
void print_queue(obj_t fifo_t q) {
    xbt_fifo_iter_t item;
    job_descriptor_t descriptor;

    fprintf(stderr, "id: %d\n", xbt_fifo_size(q));
    xbt_fifo_foreach(q, item, descriptor, job_descriptor_t) {
        fprintf(stderr, "%d %d %d %d %d\n", descriptor->id, descriptor->nodes,
            descriptor->requested_duration, descriptor->actual_duration);
    }
    printf("end\n");
}

// start job()
void start_job(job_descriptor_t jd, scheduler_bookkeeping_t bk) {
    jd->start_time = HSO_get_clock();

    // Notify the job simulator of a new job
    a_task_t task;
    task = HSO_task_create("job_start", jd, (void*) jd);
    if (HSO_task_put(task, HSO_host_get(1), PORT_START_JOB) != HSO_OK) {
        xbt_assert(0, "Error while sending a job start notification to the job simulator");
    }

    // Notify the submitter
    a_task_t task;
    int id = jd->id;
    task = HSO_task_create("job_start", id, (void*) id);
    if (HSO_task_put(task, jd->submitter, jd->channel_queue_size) != HSO_OK) {
        xbt_assert(0, "Error while sending a job start notification");
    }

    // Notify the simulator of my queue size
    // (this could really be done anywhere)
    a_task_t task;
    int queue_size = xbt_fifo_size(bk->queued);
    task = HSO_task_create("queue_size", id, (void*) queue_size);
    if (HSO_task_put(task, jd->submitter, jd->channel_queue_size) != HSO_OK) {
        xbt_assert(0, "Error while sending my queue size");
    }

    return;
}

// scheduler_init()
void scheduler_init(job_scheduler_algorithm_t alg, scheduler_bookkeeping_t bk) {
    // Initialize the job descriptor queues and the number of free nodes
    bk->queued = xbt_fifo_new();
    bk->running = xbt_fifo_new();

    // HSO-specific initialization
    switch (alg) {
        case PCFS:
            bk->scheduler_init_pcfs(bk);
    }
}

```

```
Terminal - vim -- 98x69
// find target schedulers()
void find_target_schedulers(int s, scheduler_info_t preferred,
                          scheduler_info_t* schedulers, int num_schedulers,
                          int num_target_schedulers, scheduler_info_t** target_schedulers) {
    int i;
    int r;
    int done;

    *num_target_schedulers = 0;
    for (i = 0; i < num_schedulers; i++) {
        if (i == 0) { // First, pick the preferred scheduler
            for (j = 0; j < num_schedulers; j++) {
                if (schedulers[j] == preferred) {
                    r = j;
                    break;
                }
            }
        } else {
            while (1) {
                r = random_integer_bounded(0, num_schedulers - 1);
                if (schedulers[r] == preferred) {
                    break;
                }
            }
        }
        done = 1;
        for (j = i; j < i + 1; j++) {
            if (schedulers[j] == schedulers[r]) {
                done = 0;
                break;
            }
        }
        (*num_target_schedulers)++;
        *target_schedulers = REALLOC(*target_schedulers,
                                     num_target_schedulers + 1, scheduler_info_t);
        (*target_schedulers)[(*num_target_schedulers - 1)] = schedulers[r];
    }
    return;
}

// find all target schedulers()
void find_all_target_schedulers(int s, scheduler_info_t preferred,
                              scheduler_info_t* schedulers, int num_schedulers,
                              int num_target_schedulers, scheduler_info_t** target_schedulers) {
    scheduler_info_t* sorted;
    int i;
}

```

```
Terminal - vim -- 96x42
#include "simulator.h"
#include "stdlib.h"

// Receiver function
// arg #1: port
// arg #2: dynar name
// arg #3: arg
int receiver(int argc, char** argv) {
    int port = 1;
    xbt_fifo_t fifo;
    dynar_t dynar;

    // Process the first argument
    if (strcmp(argv[1], "id_sport") != 1) {
        xbt_assert(0, "Invalid port: %s for a receiver process", argv[1]);
    }

    // Get the dynar
    if (!dynar = (dynar_t) btdict_get_or_null(receiver_dynar_dict, (const char*) argv[2])) {
        xbt_assert(0, "Cannot find dynar '%s' for a receiver process", argv[2]);
    }

    // Main loop
    while (1) {
        int r;
        a_task_t task = NULL;

        if (HSO_task_get(&task, port) != HSO_OK) {
            xbt_assert(0, "Error while receiving a task in a receiver process");
        }

        // Put the task at a random location in the dynar
        r = random_integer(0, HSO_G_dynar_length(dynar) - 1);
        dynar_insert_at(dynar, dynar_length(dynar), (void*) task);

        return;
    }
}

```

Concurrent Programs

- A sequential program does this

```
Terminal - vim - 98x69
// submit request
// Low-level function called by most of the functions below
void submit_request(a_host_t scheduler_host, int id, int nodes,
double actual_duration, double requested_duration,
int port_started, int port_done, int port_queue_size) {
job_request_t job_request; // for communication with the scheduler
a_task_t task;

job_request = (job_request_t) calloc(1, sizeof(_job_request_t));
job_request->id = id;
job_request->nodes = nodes;
job_request->actual_duration = actual_duration;
job_request->requested_duration = requested_duration;
job_request->channel_started = port_started;
job_request->channel_done = port_done;
job_request->channel_queue_size = port_queue_size;

task = HSD_task_create("request", id, (void*) job_request);
if (HSD_task_put(task, scheduler_host, PORT_HSD_REQUEST) != HSD_OK) {
xtb_assert(0, "Error while sending to scheduler. Is on channel PORT_HSD_REQUEST, HSD_host_t
scheduler_host");
}

// submit job()
// submit the request to scheduler(s)
void submit_job(pending_job_t pending_job,
scheduler_info_t scheduler_info, int num_schedulers,
scheduler_info_t *target_schedulers,
across_cluster_percentages_t the_t_across_cluster_percentages,
intra_cluster_percentages_t the_intra_cluster_percentages,
int port_started, int port_done, int port_queue_size) {
int i;
int num_target_schedulers =
scheduler_info->target_schedulers;
// Find out the list of appropriate schedulers
find_target_schedulers(num_schedulers, num_schedulers,
pending_job,
across_cluster_percentages,
target_schedulers, &target_schedulers);
xtb_assert(0, "No target scheduler found for job()");
// Send requests to each target scheduler
for (i = 0; i < num_target_schedulers; i++) {
submit_job_to_target_scheduler(target_schedulers[i], pending_job,
across_cluster_percentages,
port_started, port_done, port_queue_size);
}
free(target_schedulers);
}

// submit job to target scheduler()
// With possible multiple requests to each
void submit_job_to_target_scheduler(scheduler_info_t scheduler_info, pending_job_t pending_job,
intra_cluster_percentages_t the_intra_cluster_percentages,
int port_started, int port_done, int port_queue_size)
```

```
Terminal - vim - 101x71
// print queue()
void print_queue(xbt_fifo_t q) {
xbt_fifo_iter_t item;
job_descriptor_t descriptor;
printf("Queue contents: %d\n", xbt_fifo_size(q));
xbt_fifo_foreach(q, descriptor, job_descriptor_t) {
printf("%d %d %d %d %d %d %d %d %d\n", descriptor->id, descriptor->nodes,
descriptor->requested_duration, descriptor->actual_duration);
}
printf("\n");
}

// start job()
void start_job(job_descriptor_t jd, scheduler_bookkeeping_t *bk) {
// start time = HSD_get_clock();
// Notify the job simulator of a new job
a_task_t task;
task = HSD_task_create("job start", jd, (void*) jd);
if (HSD_task_put(task, HSD_host_self(), PORT_START_JOB) != HSD_OK) {
xtb_assert(0, "Error while sending a job start notification");
}

// Notify the simulator of my queue size
// (this could really be done anywhere)
a_task_t task;
int queue_size = xbt_fifo_size(bk->queue);
task = HSD_task_create("queue size", jd, (void*) queue_size);
if (HSD_task_put(task, jd->submitter, jd->channel_queue_size) != HSD_OK) {
xtb_assert(0, "Error while sending a job queue notification");
}

return;
}

// scheduler init()
void scheduler_init(job_descriptor_t jd, sig, scheduler_bookkeeping_t *bk) {
// initialize the job descriptor queues and the number of free nodes
bk->queue = xbt_fifo_create();
bk->running = xbt_fifo_create();

// specific initialization
case PCFS:
scheduler_init_pcfs(bk);
}
```

```
Terminal - vim - 98x69
break;
}

if ((num_target_schedulers > 1)
pending_job->flooded_across = 1;
free(pruned_schedulers);
return;
}

// find job target schedulers()
void find_job_target_schedulers(int *s, scheduler_info_t preferred,
scheduler_info_t *schedulers, int num_schedulers,
int num_target_schedulers, scheduler_info_t **target_schedulers) {
int i;
int r;
int done;

num_target_schedulers = 0;
// CHOOSE THE TARGET SCHEDULERS
for (i = 0; i < num_schedulers; i++) {
if (i < num_target_schedulers) {
// First, pick the preferred scheduler
if (schedulers[i] == preferred) {
r = i;
} else {
// If not preferred, choose randomly
r = random_integer_bounded(0, num_schedulers-1);
}
} else {
// If done, choose randomly
r = random_integer(0, num_schedulers-1);
}
done = 1;
for (j = i; j < i+1; j++) {
if (schedulers[j] == preferred) {
done = 0;
}
}
}
}

num_target_schedulers++;
target_schedulers = REALLOC(target_schedulers,
num_target_schedulers, scheduler_info_t *);
target_schedulers[num_target_schedulers-1] = schedulers[r];
}

// find scheduler target schedulers()
void find_scheduler_target_schedulers(int *s, scheduler_info_t preferred,
scheduler_info_t *schedulers, int num_schedulers,
int num_target_schedulers, scheduler_info_t **target_schedulers) {
scheduler_info_t *sorted;
int i;
}
```

```
Terminal - vim - 96x42
#include "simulator.h"
#include "stdlib.h"

// Receiver function
// arg #1: port
// arg #2: dynar name
// arg #3: argv[]
int receiver(int argc, char *argv[]) {
int port = atoi(argv[1]);
xbt_fifo_t fifo;
dynar_t dynar;

// Process the first argument
if (strcmp(argv[1], "fd", port) != 1) {
xtb_assert(0, "Invalid port. Is for a receiver process, argv[1]");
}

// Get the dynar
// If ((dynar = (dynar_t) xbt_dynar_get_dynar_null(receiver_dynar_dict, (const char *) argv[2])) {
xtb_assert(0, "Cannot find dynar 'Is' for a receiver process, argv[2]");
}

// Main loop
while (1) {
int r;
a_task_t task = NULL;
if (HSD_task_get(&task, port) != HSD_OK) {
xtb_assert(0, "Error while receiving a task in a receiver process");
}

// Put the task at a random location in the dynar
r = random_integer(0, HSD_G_dynar_length(dynar)-1);
dynar_insert_at(dynar, dynar_length(dynar), (void*) task);
}

return;
}
```


Concurrent Programs

■ A concurrent program does this

- a blue task
- a red task

```

void submit_request(struct scheduler_host, int id, int nodes,
double actual_duration, double requested_duration,
int port_queue_size) {
  // Submit a request for a job to the scheduler.
  // job request
  struct job_request {
    int job_id;
    double actual_duration;
    double requested_duration;
    int port_queue_size;
    int port_started;
    int port_done;
    int port_queue_size;
  };
  struct job_request req;
  req.job_id = id;
  req.actual_duration = actual_duration;
  req.requested_duration = requested_duration;
  req.port_queue_size = port_queue_size;
  req.port_started = 0;
  req.port_done = 0;

  // Submit a request to the scheduler.
  int req_id = xbt_fifo_push_back(queue, &req);
  req.job_id = id;
  req.actual_duration = actual_duration;
  req.requested_duration = requested_duration;
  req.port_queue_size = port_queue_size;
  req.port_started = 0;
  req.port_done = 0;

  // Submit a request to the scheduler.
  int req_id = xbt_fifo_push_back(queue, &req);
  req.job_id = id;
  req.actual_duration = actual_duration;
  req.requested_duration = requested_duration;
  req.port_queue_size = port_queue_size;
  req.port_started = 0;
  req.port_done = 0;

  // Submit a request to the scheduler.
  int req_id = xbt_fifo_push_back(queue, &req);
  req.job_id = id;
  req.actual_duration = actual_duration;
  req.requested_duration = requested_duration;
  req.port_queue_size = port_queue_size;
  req.port_started = 0;
  req.port_done = 0;
}
  
```

```

void start_job(struct job_descriptor *jd, struct scheduler_bookkeeping *bk) {
  // Start a new job.
  // Notify the simulator of a new job.
  struct task {
    int id;
    int job_id;
    int queue_size;
    int task_id;
  };
  struct task task;
  task.id = jd->id;
  task.job_id = jd->job_id;
  task.queue_size = xbt_fifo_size(queue);
  task.task_id = xbt_fifo_index(queue, jd->job_id);

  // Notify the simulator of a new job.
  struct task task;
  task.id = jd->id;
  task.job_id = jd->job_id;
  task.queue_size = xbt_fifo_size(queue);
  task.task_id = xbt_fifo_index(queue, jd->job_id);

  // Notify the simulator of a new job.
  struct task task;
  task.id = jd->id;
  task.job_id = jd->job_id;
  task.queue_size = xbt_fifo_size(queue);
  task.task_id = xbt_fifo_index(queue, jd->job_id);
}
  
```

```

break;
}

if (#num_target_schedulers > 1)
  pending_job->floored_across = 1;
free(pruned_schedulers);
return;
}

void find_job_target_schedulers(int job_id, struct scheduler_info_t preferred,
struct scheduler_info_t *schedulers, int num_schedulers,
int num_target_schedulers, scheduler_info_t **target_schedulers) {
  // Find a target scheduler for the job.
  // If there are multiple target schedulers, choose one randomly.
  // If there is only one target scheduler, choose it.
  // If there are no target schedulers, return 0.
  struct scheduler_info_t *target_scheduler;
  int i;
  int j;
  int done = 0;
  int r;

  for (i = 0; i < num_schedulers; i++) {
    if (schedulers[i] == preferred) {
      target_scheduler = schedulers[i];
      num_target_schedulers++;
    }
  }

  if (num_target_schedulers == 1) {
    target_scheduler = schedulers[0];
  } else {
    r = random_integer_bounded(0, num_schedulers - 1);
    target_scheduler = schedulers[r];
  }

  if (done)
    break;
}

return;
}

void find_job_target_schedulers(int job_id, struct scheduler_info_t preferred,
struct scheduler_info_t *schedulers, int num_schedulers,
int num_target_schedulers, scheduler_info_t **target_schedulers) {
  // Find a target scheduler for the job.
  // If there are multiple target schedulers, choose one randomly.
  // If there is only one target scheduler, choose it.
  // If there are no target schedulers, return 0.
  struct scheduler_info_t *target_scheduler;
  int i;
  int j;
  int done = 0;
  int r;

  for (i = 0; i < num_schedulers; i++) {
    if (schedulers[i] == preferred) {
      target_scheduler = schedulers[i];
      num_target_schedulers++;
    }
  }

  if (num_target_schedulers == 1) {
    target_scheduler = schedulers[0];
  } else {
    r = random_integer_bounded(0, num_schedulers - 1);
    target_scheduler = schedulers[r];
  }

  if (done)
    break;
}

return;
}
  
```

```

#include "simulator.h"
#include "stdlib.h"

// Receiver function
// arg #1: port
// arg #2: dynar name
//
int receiver(int argc, char **argv) {
  // Get the port.
  int port = atoi(argv[1]);

  // Process the first argument.
  // If (strcmp(argv[1], "id") == 0) {
    xbt_assert(0, "Invalid port '%s' for a receiver process", argv[1]);
  }

  // Get the dynar.
  // If ((dynar = (dynar_t)xbt_dict_get_dynar_null(receiver_dynar_dict, (const char *)argv[1])) {
    xbt_assert(0, "Cannot find dynar '%s' for a receiver process", argv[1]);
  }

  // Main loop.
  while (1) {
    int r;
    struct task task = NULL;

    if (!xbt_fifo_get(task, port) != MSO_OK) {
      xbt_assert(0, "Error while receiving a task in a receiver process");
    }

    // Put the task at a random location in the dynar.
    r = random_integer(0, xbt_fifo_length(dynar) - 1);
    dynar_insert_at(dynar, dynar_length(dynar), (void*)task);
  }

  return 0;
}
  
```


Concurrent Programs

- Thinking about what a concurrent program does is more difficult than for a sequential program
 - One may have to keep a mental picture of what each task is doing at all time (we try not to)
 - Questions like “While task #1 is in function f where is task #2?” are often difficult (and we try not to have their answers matter so that we don’t have to ask them)
 - Two executions of the same program may not be identical
 - We’ll explain this in more details
- As a result, concurrent programs are
 - Almost always more difficult to design for correctness
 - Almost always more difficult to read
 - Always more difficult to debug
- So, why do we bother at all?

Concurrency for Interactivity

- One of the oldest uses for concurrency is to make programs more interactive
- While a program is running and doing stuff, the user should still be able to interact with it
- Example:
 - What if in your Web browser you couldn't click "back" before the browser has finished loading the page you immediately realized is the wrong one?
 - What if in iTunes you couldn't look at your play list while you're playing a song because the program is busy playing the song?
- One wants to avoid the "frozen because I am working" problem as much as possible
- Let's look at a made-up example...

Designing a Concurrent GUI

- A common application of concurrent programming is for designing Graphical User Interfaces (GUIs)
- Example application
 - Say you want to write a program that renders 3-D objects on the screen
 - You have a clickable button to launch the rendering
 - But rendering takes a long time
 - You don't want the GUI to appear "frozen" while rendering the objects
 - For instance, you want the "Quit" and "Cancel" buttons to still work

Concurrent GUI?

- One way to avoid the “frozen” problem **without using concurrency** is to write your code with **breaking down a task into sub-tasks**
 - Typically, I’ll write code fragments in C/C++-like pseudo-code or in Java, without declarations, etc.

```
void render(...) {
    for (step=0; step<100; step++) {
        this.doSomeRendering(...);
        if (gui.cancelButton.clicked())
            break;
    }
}
```

That was (often) a bad idea

- It's cumbersome:
 - What if you want to do 7 tasks?
 - Sprinkle “interaction checks” throughout your code will make is unreadable and annoying to maintain
- It's no always doable:
 - What if rendering is not breakable into multiple calls??
 - Perhaps you call some library that you didn't write and cannot modify
- What if some tasks have some real-time requirements?
 - e.g., you want to have an animated symbol that changes every t milliseconds but a call to `doSimpleRendering()` takes longer than that?



Task-based design

- Instead of thinking of your application as one task that has to juggle many things at once, you think of your application as a bunch of tasks that run concurrently
- Each task does one thing and perhaps doesn't even know that there are other tasks
- Assuming that we have a programming language that allows us to define tasks we can rewrite our application...

Concurrent GUI

- In *horrible* pseudo-code:

```
renderer = new Task(render)
mousetatcher = new Task(watchmouse)
```

```
renderer.start();
mousetatcher.start();
renderer.wait_until_finished()
```

```
Renderer::render() {
    // do rendering on screen
}
```

```
Mousewatcher::watchmouse() {
    // whenever mouse clicked, kill renderer
}
```

Concurrent Tasks Abstraction

- Fortunately, our OSs support the **concurrent tasks abstraction**
 - After all, on our machines many programs can run simultaneously
 - So why not tasks within our programs?
- This can be done by
 - A special library
 - A virtual machine like the JVM
 - The Operating System
 - A combination of the above
- Almost all modern programming languages allow you to create “tasks” in your programs

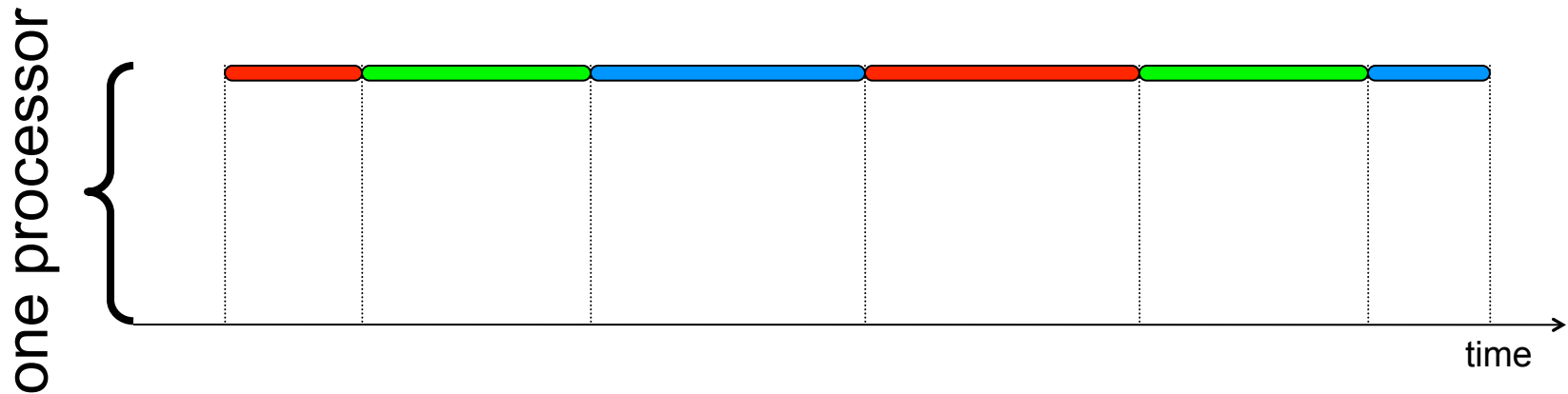
“Simultaneous” tasks?

- Can we really have simultaneous concurrent tasks?
- There are two kinds of concurrency:
 - **True concurrency**: two or more “things” happen at the same instant in time
 - **False concurrency**: only one thing happens at a time, but the illusion of concurrency is achieved because the OS performs rapid context switching

True/False Concurrency

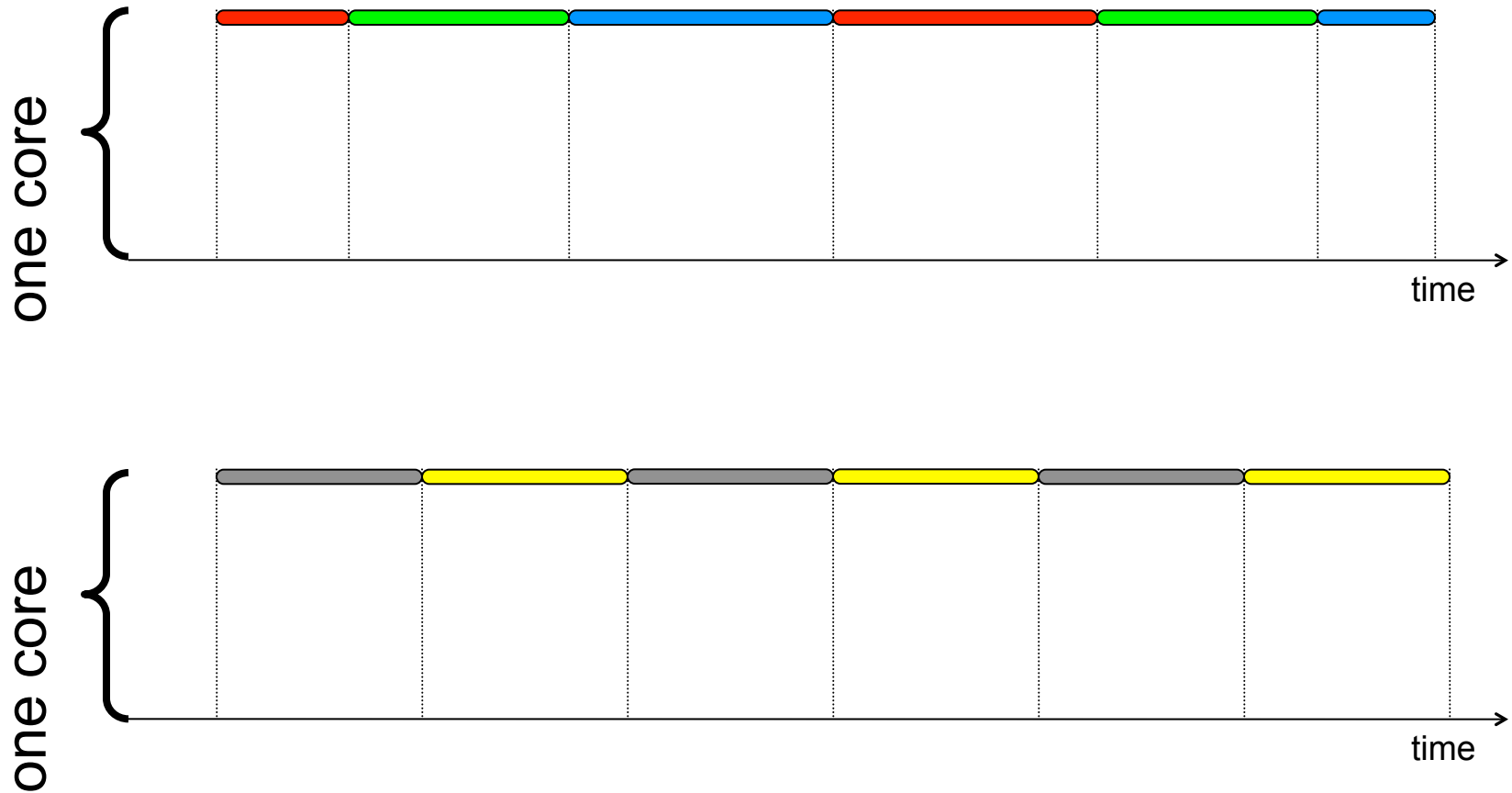
- Consider a program that defines two concurrent tasks, T1 and T2
- On a single core, only one task can use the CPU
 - The concurrent tasks use **false concurrency**
- On a multi-core system, each task can be on a different core
 - The concurrent tasks use **true concurrency**

False Concurrency on One Core



- False concurrency between the red task, the green task, and the blue task
- The OS context-switches back and forth between the three tasks
- Because this is very fast, we have the “illusion” of simultaneous execution

On Two Cores



- **True concurrency** between the yellow task and the green task, the grey and the blue, etc.

True/False Concurrency

- The programmer shouldn't have to care/know whether concurrency will be true or false
 - Besides the fact that true concurrency offers better performance than false concurrency
- Typically, the programmer doesn't know on which computer the program will run!
 - You have no idea how many cores your "customer" will have on their machines
- A concurrent program with 10 tasks will work on a single-core processor, a quad-core processor, a 32-core processor, etc
 - Your job as a developer is to create tasks
 - e.g., the program could easily discover that the machine it's running on has 8 cores, and thus decides to create 8 tasks
 - The job of the OS is to dispatch these tasks to the cores
 - e.g., the OS is smart enough to put each of the 8 tasks on its own core without you having to make those decisions

Performance!!!

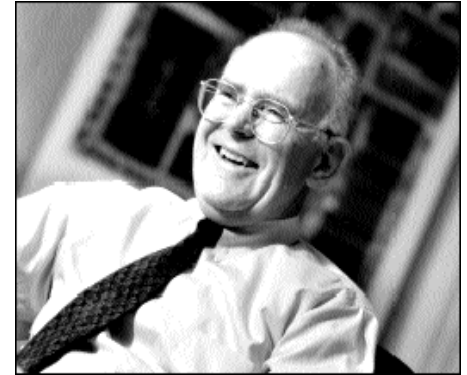
- But wait, with true concurrency we can also go faster!!!
 - If you have to bake 2 cakes and you have 1 oven it will take you 2 hours
 - But if you have 2 ovens that can be on at the same time, it will take you only 1 hour
- This brings us to the second major reason why people want to use concurrency: compute stuff faster
- To summarize we have two motivations:
 - concurrency for interactivity
 - concurrency for performance

Multi-core Processors

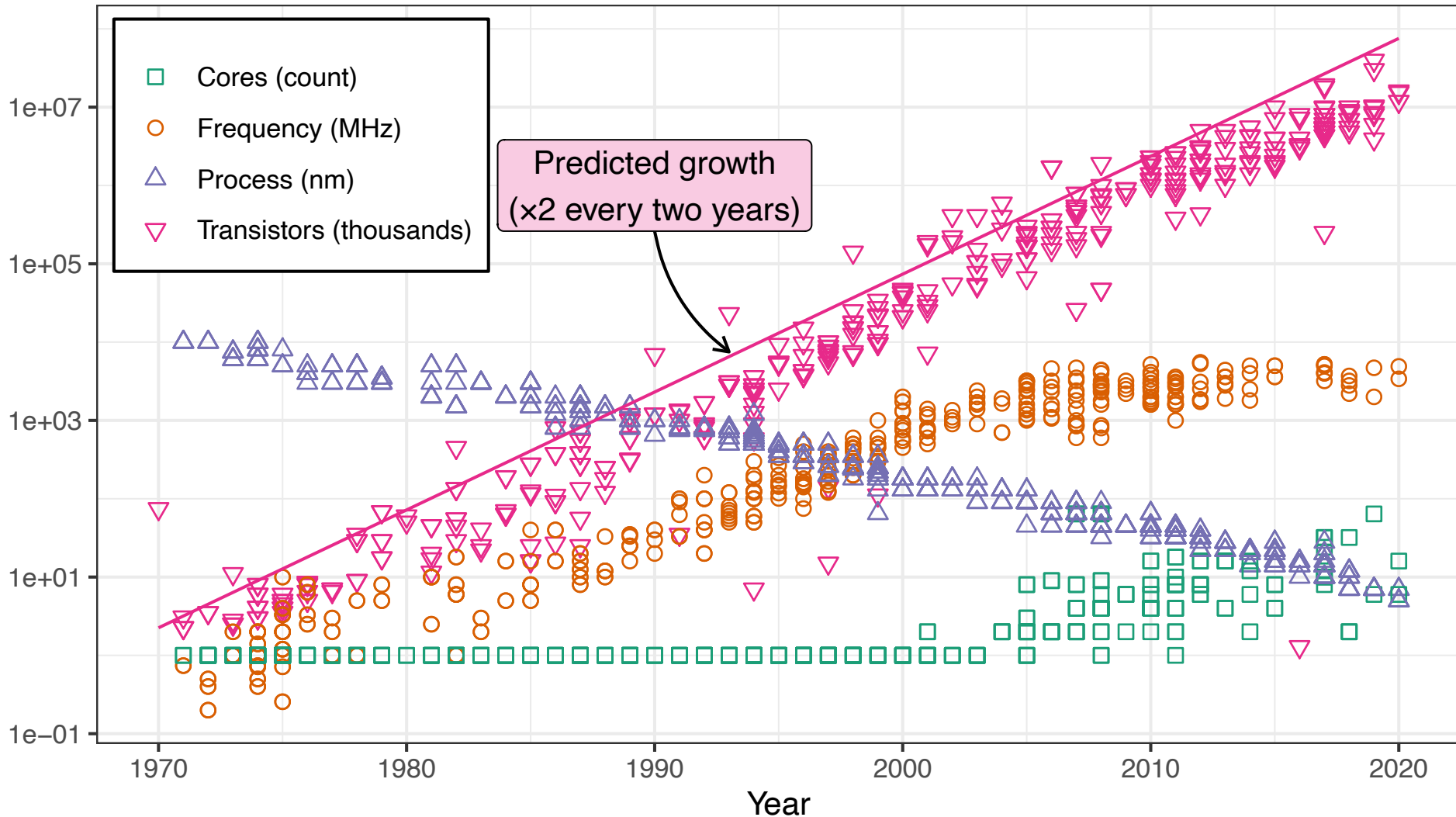
- There have always been different hardware resources to use concurrently to increase performance
 - e.g., the disk, the network, and the CPU can all be used at the same time because they are different pieces of hardware
- But the last decades have seen the advent of multi-core processors, which are now ubiquitous
- Many programs have been made concurrent so as to utilize multiple cores concurrently
 - It's become impossible to say "I am an employable software developer but I don't deal with concurrency"
- How come we have multi-core processors in the first place?

Moore's Law

- In 1965, Gordon Moore (co-founder of Intel) predicted that **transistor density** of semiconductor chips would **double** roughly **every 24 months** (often “misquoted” as 18 months)
 - He was right
 - But, the law was often wrongly interpreted as: “Computers get twice as fast every 2 years”
 - This wrong interpretation was true for a while, but no longer...

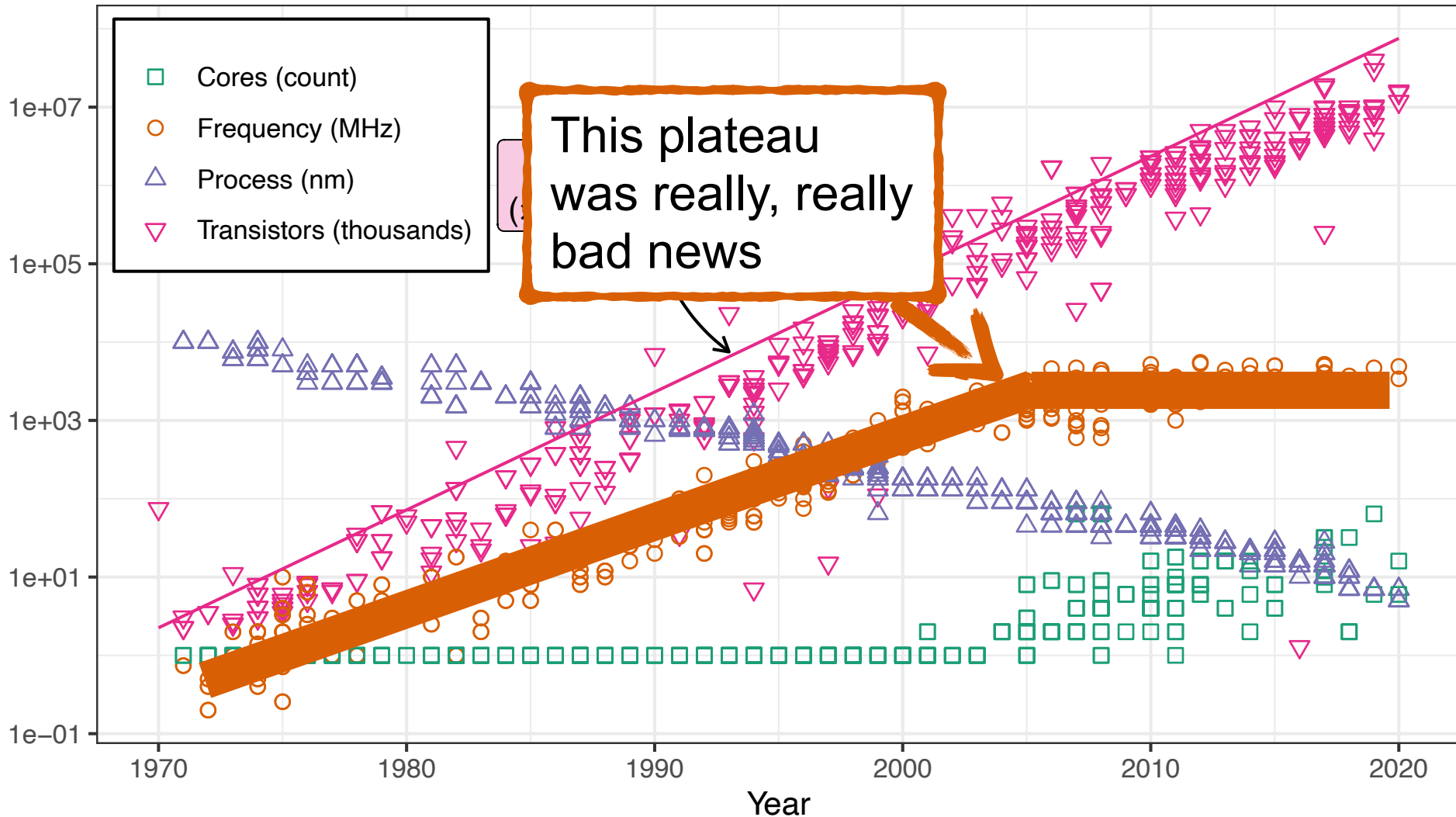


50-year Trend



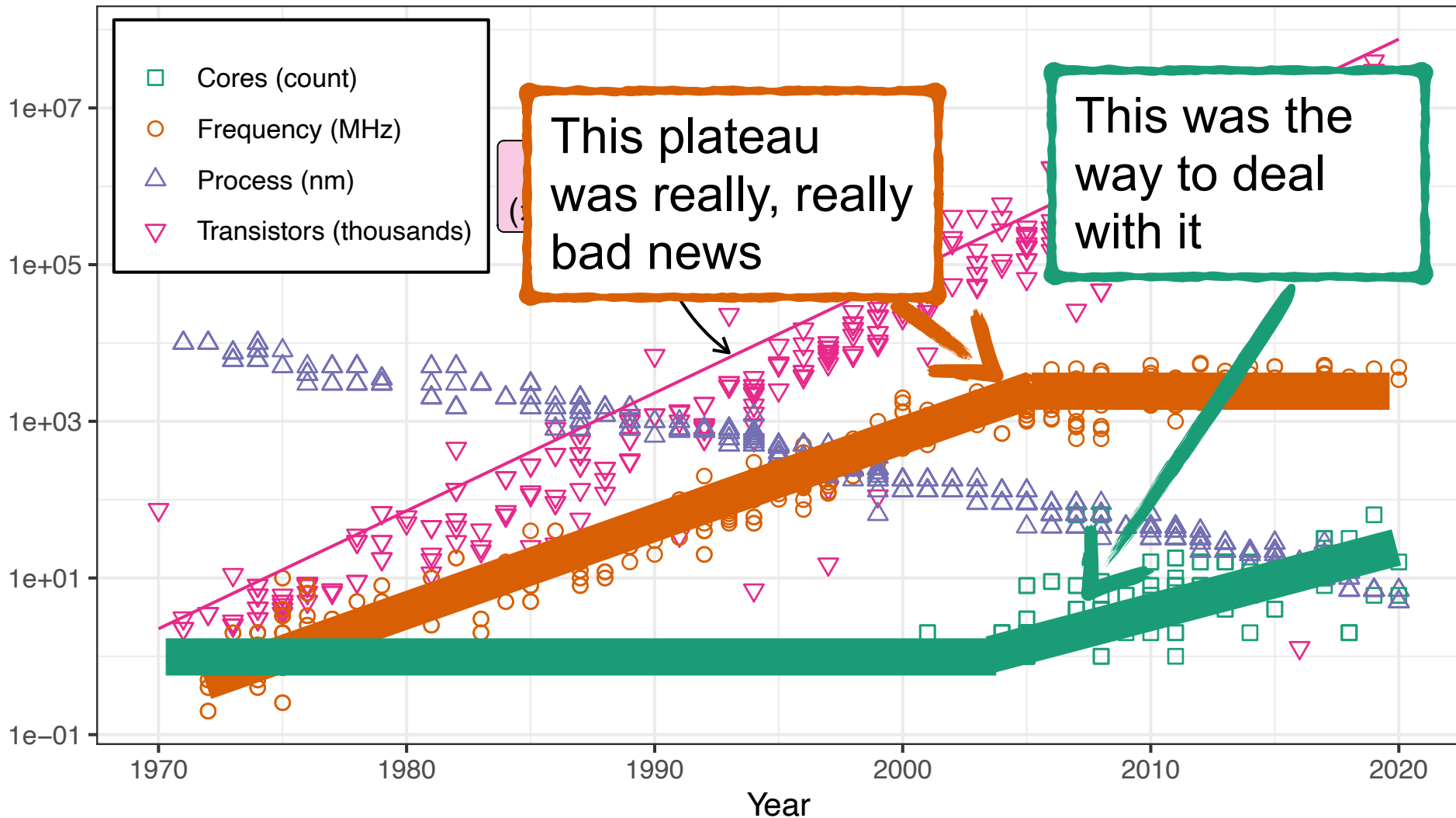
Plot inspired from the work of Pedro Bruel, generated with data from Wikipedia [Wik21a; Wik21b].

50-year Trend



Plot inspired from the work of Pedro Bruel, generated with data from Wikipedia [Wik21a; Wik21b].

50-year Trend



Plot inspired from the work of Pedro Buel, generated with data from Wikipedia [Wik21a; Wik21b].

Multi-core Chips

- Constructors cannot increase clock rate further
 - Power/heat issues
- They bring you **multi-core processors**
 - Multiple “low” clock rate processors on a chip
- It’s really a solution to a problem, not a cool new advance
 - Even though there are many cool/interesting things about multicore processors
 - Even though writing concurrent code is cool/interesting, as we’ll see in this course
- But most users/programmers would rather have a 100GHz core than 50 2GHz cores
 - In which case we would not need to write concurrent programs
 - When given the choice, if you can get by without concurrency, you’re likely better off (until you can’t avoid concurrency anymore)
 - i.e., in general no compiler will nicely take your sequential app and magically transform it into an efficient multi-threaded app

So, Multi-Core = High Performance?

- A big question is: **how much performance benefit can we really get from concurrency?**
- It's a difficult question because the answer
 - depends on the application
 - depends on the computer
 - depends on the language / operating system
- In some cases, it's very easy to achieve great performance via concurrency
- In others, it's very difficult

- We'll be exposed to this in this course

Take-away

- Concurrency is about structuring your programs as sets of tasks
- Typically done for interactivity and/or for performance
- Concurrency is supported by programming languages, by OSes, and by the hardware
- Issues for programmers:
 - Correctness (we'll see this can be a tough one)
 - Performance (sometimes easy, sometimes not)

Task-based Thinking

- From now on, you should begin writing code with concurrency in mind (even if the code is not concurrent right away)
- You currently think of your programs as sets of classes/objects
 - Or data structures and functions
- But now, you also have to think of your programs as sets of tasks
 - And these tasks operate on the objects
- This requires a little bit of mental adjustment, and our programming assignments will help making that adjustment
 - You have been exposed to this a little bit in ICS332, which will help too
- Thinking of programs as “sets of tasks” will become second nature soon enough
 - And is very natural for many applications actually
 - e.g., each tab in your Web browser is a task
- When I write a sequential program, I typically think of it as a concurrent program that happens to have a single task

Conclusion

- In the next set of lecture notes, we'll look at processes and threads
 - What they are (super quick ICS332 “review”)
 - How we can use them to make an application concurrent (beyond ICS332)