

# Notions of Scheduling

## ICS432 - Fall 2008 Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

## Easy Parallelization

- We have talked about the use of threads to partition work among multiple threads
- Example:
  - Some array
  - Each thread deals with some part of the array
- Therefore, if we have 4 cores, we start 4 threads, and each thread gets 1/4 of the array
- This is the easiest case for application parallelization
- **Identical work units**
  - Processing of each array element takes the same time
- **Independent work units**
  - We don't care in which order element arrays are processed
- Let's look at what happens when things are not so easy

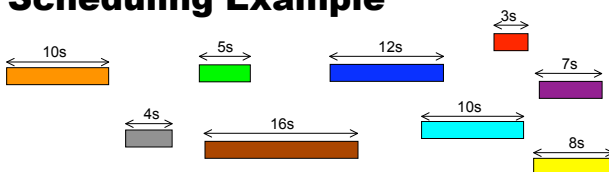
## Non-identical Work Units

- Let's say your application consists of N independent work units
  - e.g., compute N matrix inversions, like needed for instance in pattern-recognition algorithms used in computer vision, etc.
- Let's say your work units all have different computational costs and you know how long each of them takes
  - e.g., parse N biological sequences looking for the ACTGG amino-acid pattern, and the parsing time is linear in the length of each sequence, which is known
- Say we have 4 cores and we start 4 threads
- The question is: **how do you assign work units to threads?**
  - Blindly giving the first 1/4 to the 1st thread, the second 1/4 to the 2nd thread, etc. could lead to bad results
  - e.g., what if the distribution of sequence lengths is not uniform, if sequences are sorted by length, etc.
- Goal: minimize execution time

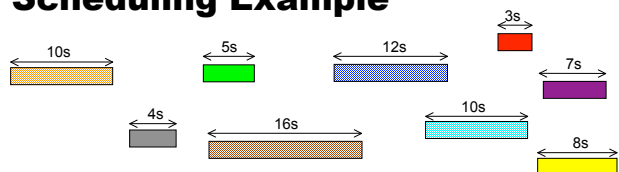
## Non-identical Work Units

- Turns out that this is a very difficult problem
- It is NP-hard
  - Trivial reduction to 2-partition for 2 processors for instance
- So one has to use some heuristic
  - Could be very complicated
- A simple heuristic:
  - Sort the work units from the longest one to the shortest one
  - For each work unit, assign it to the core that would finish it the soonest, accounting for what work units were assigned to that core already
- Let's see this on an example

## Scheduling Example



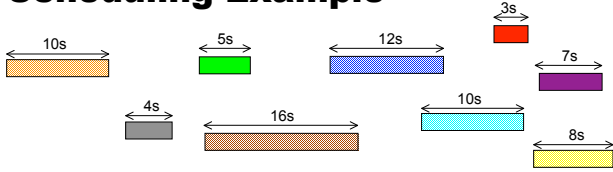
## Scheduling Example



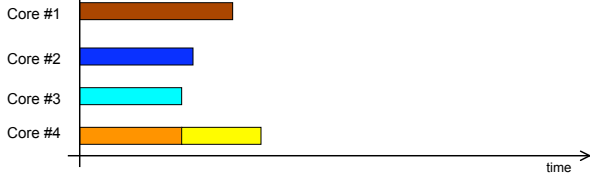
the four biggest tasks are given one to each core



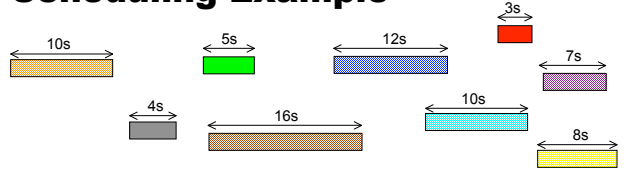
## Scheduling Example



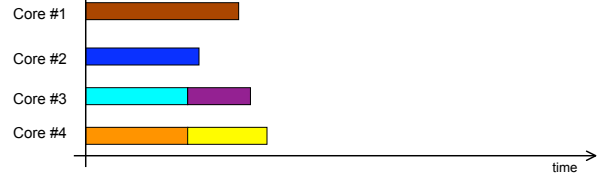
the next biggest task (8s) should go on Core #3 or #4



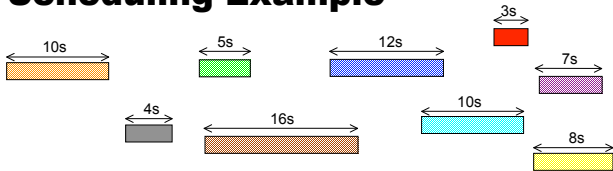
## Scheduling Example



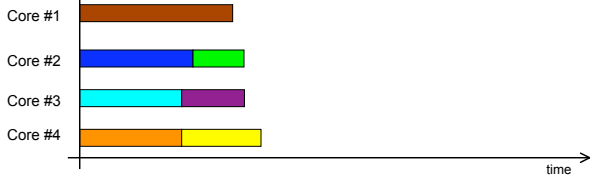
the next biggest task (7s) should go on Core #3



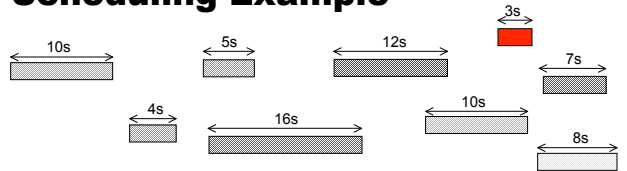
## Scheduling Example



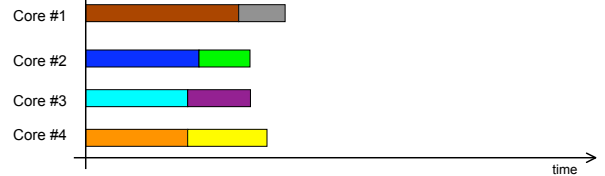
the next biggest task (5s) should go on Core #2



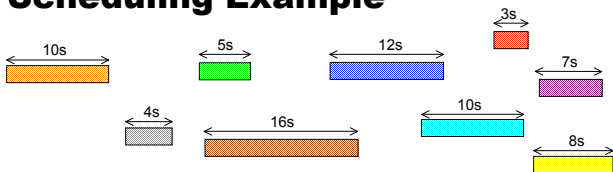
## Scheduling Example



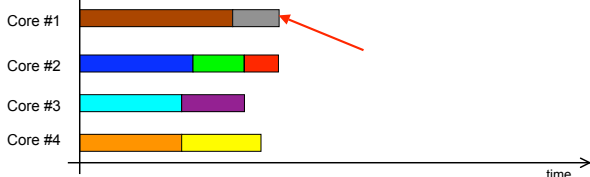
the next biggest task (4s) should go on Core #1



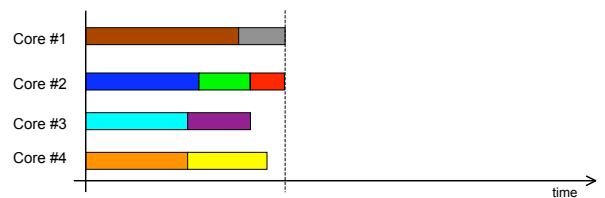
## Scheduling Example



the next biggest task (3s) should go on Core #2 or #3



## Scheduling Example



- The obtained execution time is  $16+4 = 20s$
- The obtained schedule is
  - Give tasks brown and grey to a core
  - Give tasks blue, green and red to another core
  - Give tasks cyan and purple to another core
  - Give tasks orange and yellow to another core
  - The 3rd and 4th cores will be idle for a little bit of time

## Scheduling Problem

- Definition of a Scheduling Problem
  - Given some tasks
    - computations, data transfers, tasks in a factory
  - Given some resources
    - processor cores, network links, tools
  - Assign tasks to resources in a way that some performance metric is optimized
    - minimizing overall execution time
    - maximizing network utilization
    - maximizing factory productivity
- When trying to minimize execution time, it is often the case that we want resources to finish computing at the same time (as much as possible)
  - That's what we did in the previous example
  - Called **load balancing**

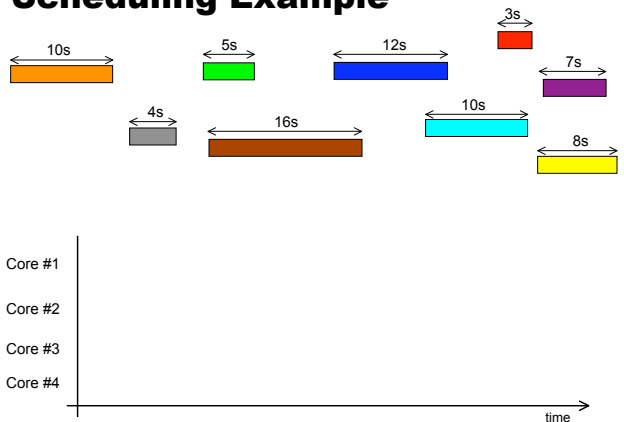
## Non-deterministic Work Units

- Let's say we still have N independent work units
- Let's say the work units are all different
- But now let's say that **we do not know how long they take!**
  - Some will be short
  - Some will be long
  - But we don't know which ones
- Example: Ray tracing
  - You have a scene to render with several objects in it
  - Ray tracing consists in shooting a photon (or "ray") through each pixels of the image, which is a window into the scene
  - The ray bounces around and then you can trace it back to figure out the color of that pixel
  - Some pixels are easy to compute
    - e.g., the ray just hits a black surface
  - Some pixels are difficult to compute
    - e.g., the ray goes through transparent material, is reflected off shiny surfaces, etc.
  - Unless you spend a lot of time analyzing the scene beforehand, you don't really know (or at least your program doesn't really know) which pixels will be simple and which pixels will be complicated

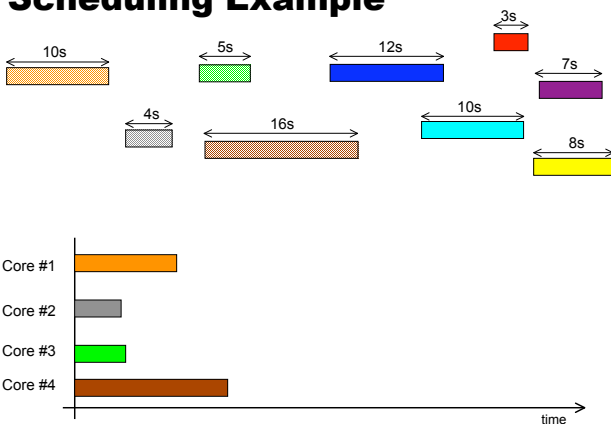
## Greedy Scheduling

- The question is: if work units are non-deterministic, which ones do we assign to which core?
- The answer is: we cannot know ahead of time and we let cores "request" work units
  - Called "greedy" or "on-demand" scheduling
- We want the program to be written logically as
  - A worker:
    - ask for work, do work, repeat
  - A master:
    - wait for a request for work, assign work, repeat...
- You could implement this easily with pthreads for instance, with locks and condition variables for communication between threads
- In this way
  - A worker that was assigned a long work units will not be back asking for more work for a while
  - A worker that was assigned a short work unit will be back asking for more work early

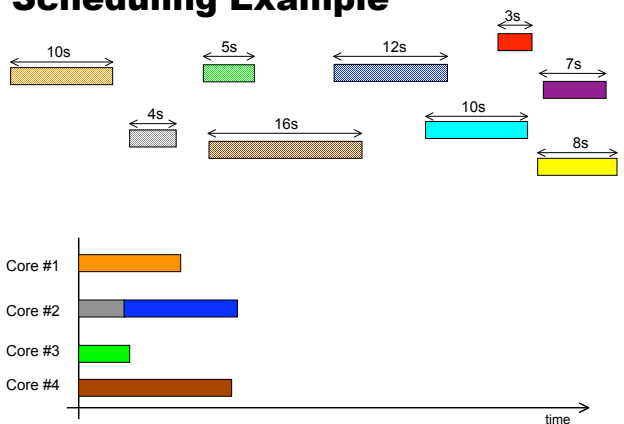
## Scheduling Example



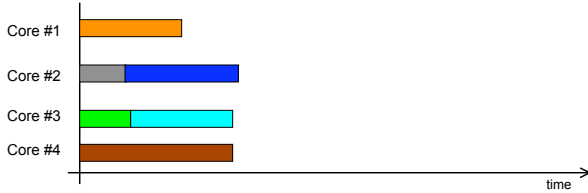
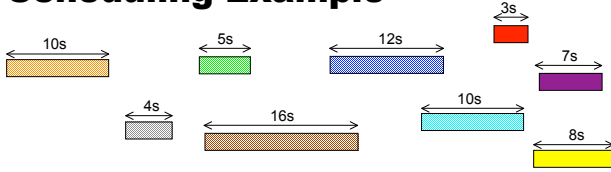
## Scheduling Example



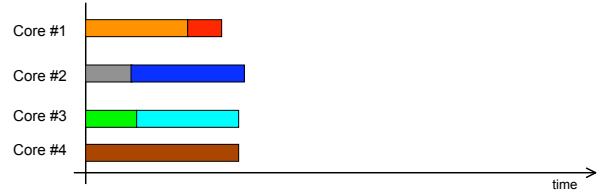
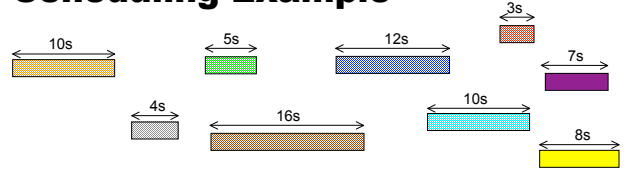
## Scheduling Example



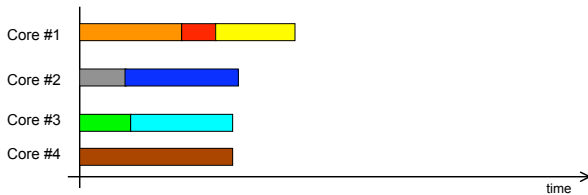
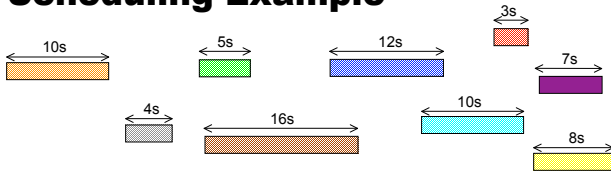
## Scheduling Example



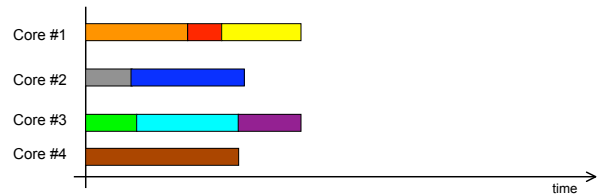
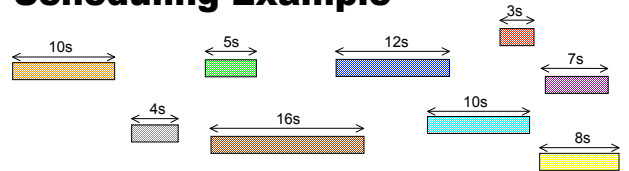
## Scheduling Example



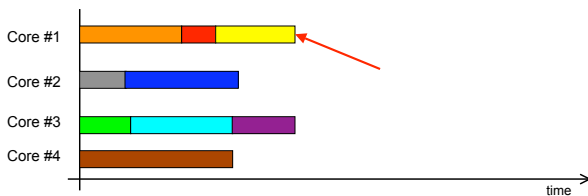
## Scheduling Example



## Scheduling Example



## Scheduling Example



- The overall execution time is  $10+3+8 = 21s$ 
  - remember that the previous one was 20s, but here we "pretended" that we didn't know task durations
- In this example, there is not much difference between the two schedules
- But things can get much worse: it's all dependent on the (hidden) distribution of task execution times

## So where are we?

- We have seen two cases for independent work units
  - I know the durations of each work unit ahead of time (whether they are identical or not)
  - I don't know these durations
- In the first case we can do **static assignment** of work units to resources
  - Initially we tell cores/threads what to do. They do it blindly. And we're done.
- In the second case we use purely **dynamic assignment** of work units to resources
  - We only know which work unit a resource computes when it asks for work

## Scheduling and Open/MP

- We can implement static or dynamic assignment of work units (also called static or dynamic scheduling) with Pthreads
  - We have done static in one of our Pthreads assignment
  - We could do dynamic in a producer-consumer fashion for instance, where the producer just produces everything at once
- Turns out, Open/MP provides very simple ways to do either type of scheduling for loops

## Open/MP Scheduling Example

```
int chunk = 3
#pragma omp parallel for \
    shared(a,b,c,chunk) private(i) \
    schedule(static,chunk)
{
    for (i=0; i < n; i++)
        c[i] = a[i] + b[i];
}
```

- Uses static scheduling
- Gives work unit by batches of 3 to threads

## Open/MP Scheduling Clauses

- When defining a parallel for loop you can specify to OpenMP whether you want static or dynamic scheduling
- You also specify a **chunk size**
- A chunk size of 2 says that OpenMP will treat 2 iterations of the loop as one work unit
  - It will give work units to threads two at a time, as opposed to one at a time
- Let's see an example

## Open/MP Scheduling Example

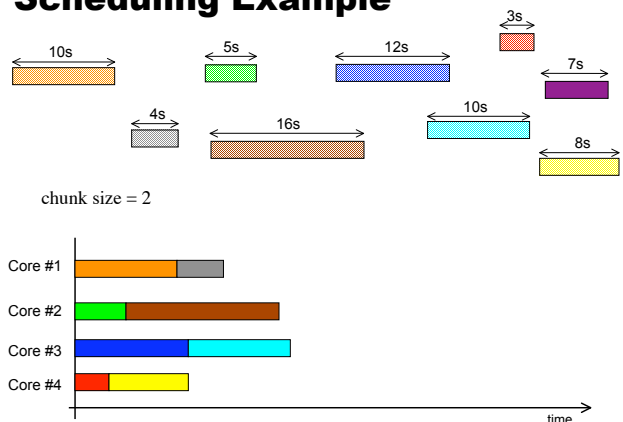
```
int chunk = 3
#pragma omp parallel for \
    shared(a,b,c,chunk) private(i) \
    schedule(dynamic,chunk)
{
    for (i=0; i < n; i++)
        c[i] = a[i] + b[i];
}
```

- Uses dynamic scheduling
- Gives work unit by batches of 3 to threads

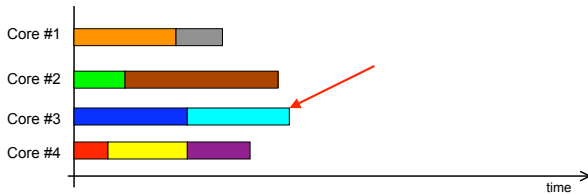
## Why a chunk size > 1?

- Why would we need to have a chunk size greater than 1 with dynamic scheduling?
- Clearly this could make the schedule worse
- Let's see this on our previous example, still pretending that we don't know the work unit execution times

## Scheduling Example



## Scheduling Example



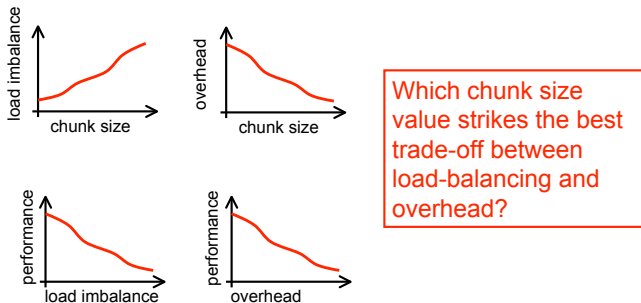
- The overall execution time is  $10+12 = 22s$ 
  - remember that the time using the heuristic was 20s, but here we "pretended" that we didn't know task durations
  - more importantly, the time with a chunk size of 1 was 21s
- Again the gap can be much larger in other examples
- With a higher chunk size, we have worse load balancing!
  - In this case Core #4 gets very little work
- So why on earth would we have a chunk size  $>1$ ???

## Scheduling and Overhead

- The problem with a chunk size of 1 and a dynamic schedule is **overhead**
- Each time a thread finishes computing a work unit, it must figure out which work unit it should do next
  - May involve locking a mutex lock, checking and updating some counter, unlocking a mutex lock
- So, if you have 1 million iterations to your loop, you're going to a 1 million lock-check-update-unlock operations
- This can end up being very costly, especially if the computation in each iteration is small
- So with a chunk size of 2, you'd do only 1/2 million lock-check-update-unlock operations
- With a chunk size of 1000, you'd do only 1000 lock-check-update-unlock operations, thereby saving many cycles

## Scheduling Conundrum

- We now face a difficult situation



## Best Chunk Size

- Determining the best chunk size is difficult
  - It depends on the statistical distribution of work unit durations, which may not be known
  - It depends on the overhead on the target computer
    - Some lock implementations may be better than others
- As a user of Open/MP, you're left making some guesses
  - A value of 1 is probably not good
  - A value of 10000000 is probably not good
  - Let's try a few, measure performance, and see what the deal is
- This is ultimately unsatisfying, and people have tried to come up with solutions that could work well no matter what the underlying tasks look like!
- One of these solutions is called **guided scheduling**

## Guided Scheduling

- The reason why we want a chunk size of 1 is that **at the end of the execution** we want to have fine-grain dynamic assignment of tasks to resources
  - you don't want to give 10 work units to the last thread is another thread is about to finish as well
  - you'd rather given them 5 work units each
  - And in fact, a chunk size of 1 is best
- The reason why we don't want a chunk size of 1 **at the beginning of the execution** is that we can live with coarse-grain dynamic assignment of tasks to resources
  - In the beginning, if there are many iterations, it's ok to let some resources be overwhelmed with a long batch of work units, as it will have time to catch up before we get to the bitter end
  - So why pay the extra overhead of a chunk size of 1?
- **Guided Scheduling:**
  - Start with large chunk sizes
  - End with small chunk sizes

## Guided Scheduling

- Say we have N work units and P threads
- One option for guided scheduling
  - We can take the first  $N/2$  work units and partition them in P chunks
  - Take the next  $N/4$  work units, and partition them in P chunks
  - Take the next  $N/8$  work units, and partition them in P chunks
  - etc.
- Then assign the chunks to resources in a greedy fashion

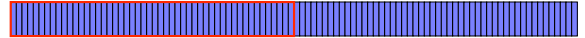
### Guided Scheduling Example

- 100 work units, 4 cores



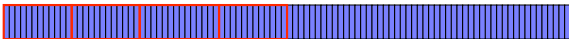
### Guided Scheduling Example

- 100 work units, 4 cores



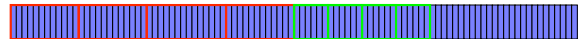
### Guided Scheduling Example

- 100 work units, 4 cores



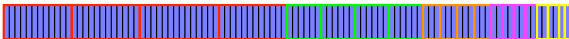
### Guided Scheduling Example

- 100 work units, 4 cores



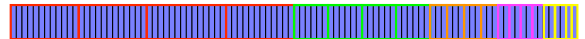
### Guided Scheduling Example

- 100 work units, 4 cores



### Guided Scheduling Example

- 100 work units, 4 cores



## Guided Scheduling Example

- 100 work units, 4 cores



- Now we have 20 chunks, workers come in in a greedy fashion and we assign chunks in order
- Some of these chunks may be shorter than "expected", some may be longer than expected
- But the point is that the yellow chunks, even if longer than expected can't be that long
  - Assumes that we don't have unbounded work unit execution times, and that things are not too "crazy", which is often the case

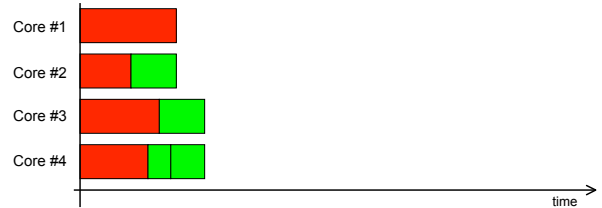
## Guided Scheduling Example



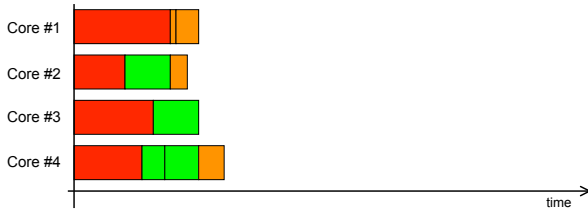
## Guided Scheduling Example



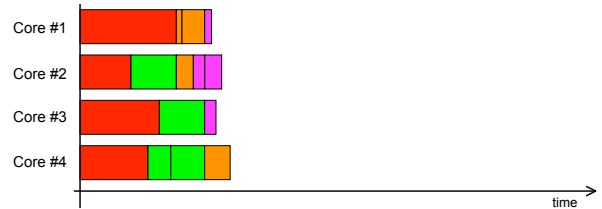
## Guided Scheduling Example



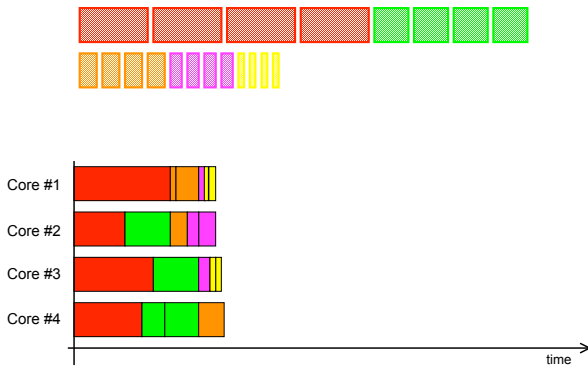
## Guided Scheduling Example



## Guided Scheduling Example



## Guided Scheduling Example



## Guided Scheduling and Open/MP

```
int chunk = 3
#pragma omp parallel for \
    shared(a,b,c,chunk) private(i) \
    schedule(guided,chunk)
{
    for (i=0; i < n; i++)
        c[i] = a[i] + b[i];
}
```

- Uses guided scheduling
- The chunk size specifies the smallest chunk size that will be used (at the end)

## Conclusion

- We have only scratched the surface of scheduling
  - It is a huge area of computer science
- Open/MP makes it easy to play with scheduling
- Doing it by hand with Pthread requires some code
  - especially for guided scheduling