Formal Representation, Scheduling, and Extracting Parallelism

Erik Saule
esaule@uncc.edu

Parallel and Distributed Programming

01/17/18

Learning Outcomes

At the end of this session you will know how to

- Give two representations of parallel codes
- Compute metrics on dependency graphs
- Interpret metrics of dependency graphs in term of parallel execution
- Represent a schedule of a parallel code on some number of processors
- Apply an algorithm to build a schedule
- Find the dependencies from a sequential code and express the dependencies as a graph

Learning Outcomes

At the end of this session you will know how to

- Give two representations of parallel codes
- Compute metrics on dependency graphs
- Interpret metrics of dependency graphs in term of parallel execution
- Represent a schedule of a parallel code on some number of processors
- Apply an algorithm to build a schedule
- Find the dependencies from a sequential code and express the dependencies as a graph

The associated assignment will show you how to

- Formulate the parallelism of simple problems as a DAG
- Extract additional parallelism from classical problem that do not seem to exhibit parallelism

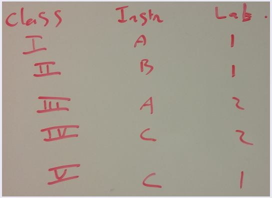
Outline

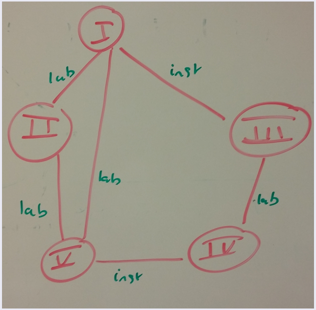
The conflict graph representation

Conflict graph

- Used to represent a set of tasks that can be executed in any order but that use a common resource.
- Undirected graph with edges that connect tasks with a conflict.
- Often, the problem to solve is to color the vertices with different colors so that two neighboring vertices have a different color.
- NP-Complete problem but greedy heuristics are good.

Example





The conflict graph representation

Conflict graph

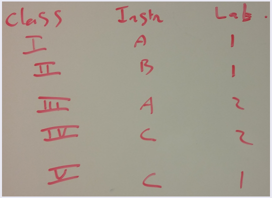
Used to represent a set of tasks that can be executed in any order but that use a common resource.

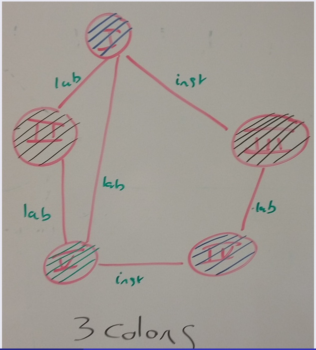
Undirected graph with edges that connect tasks with a conflict.

Often, the problem to solve is to color the vertices with different colors so that two neighboring vertices have a different color.

NP-Complete problem but greedy heuristics are good.

Example





The dependency graph representation

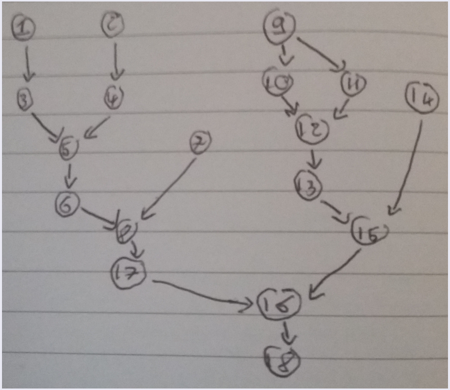
DAG representation

Represents tasks as vertices.

Represents x before y using a $x \to y$ directed edge.

The graph is always without cycles.

Example



The dependency graph representation

DAG representation

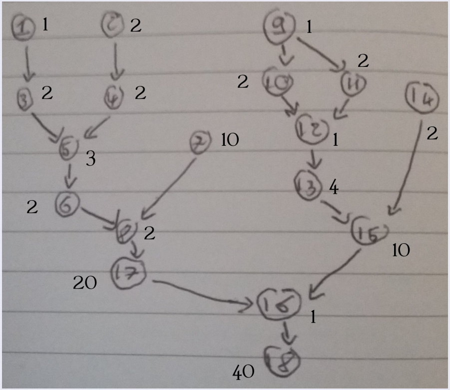
Represents tasks as vertices.

Represents x before y using a $x \rightarrow y$ directed edge.

The graph is always without cycles.

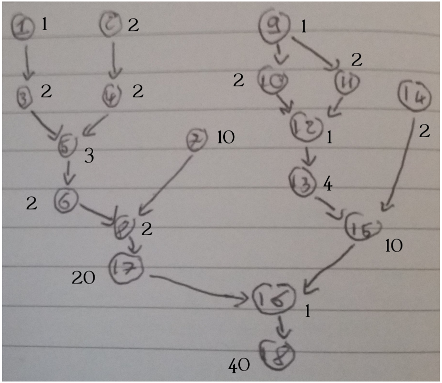
Processing time required associated with vertices, often denoted p_i

Example



A lemon pie recipe

(1) break 2 eggs and split the white and yoke
(2) cut 125g of butter in cubes
(3) mix yoke and 70g of sugar+5cl of water
(4) mix 250g of flour with butter
(5) mix (3) and (4) and make a ball
(6) spread (5)
(7) heat oven to 180C
(8) put crust (6) in pie pan
(9) wash 4 lemons
(10) peel two lemons from (9) and finely cut them
(11) press the four lemons from (9) and (10)
(12) mix lemons(11), peel(10, 160g of sugar, 1 sp of flour
(13) cook slowly (12)
(14) whip 3 eggs
(15) mix (14) and (13) and cook fast whipping
(16) empty (15) in (17)
(17) cook (8) for 20 minutes
(18) wait until (16) cools



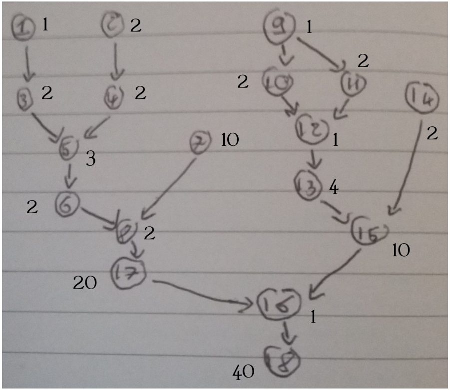
Metrics: Work

Work

Total amount of work that is to perform on the application.

Simply the sum of all processing times.

Often denoted $\sum p_i$

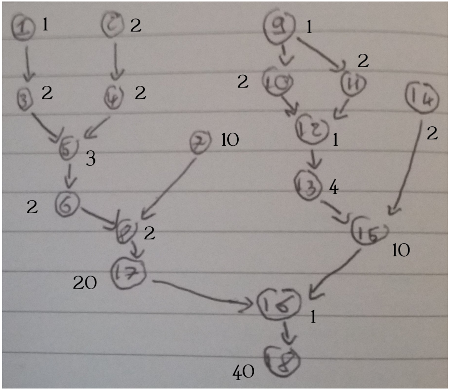


Work

Total amount of work that is to perform on the application.

Simply the sum of all processing times.

Often denoted $\sum p_i$



Here $\sum p_i = 107$

Metrics: Work

Work

Total amount of work that is to perform on the application.

Simply the sum of all processing times.

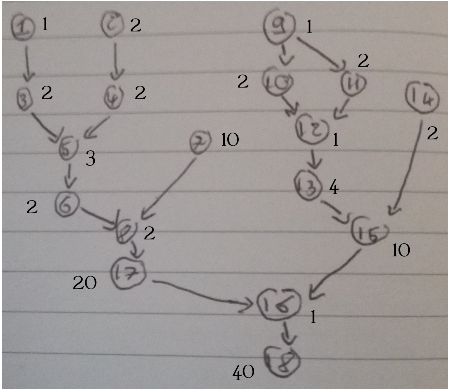
Often denoted $\sum p_i$

Usage

On P processors, the application can not be processed faster than $\frac{\sum p_i}{P}$.

$\frac{\sum p_i}{P}$ is a **lower bound** of the **makespan**.

$C_{max} \geq \frac{\sum p_i}{P}$



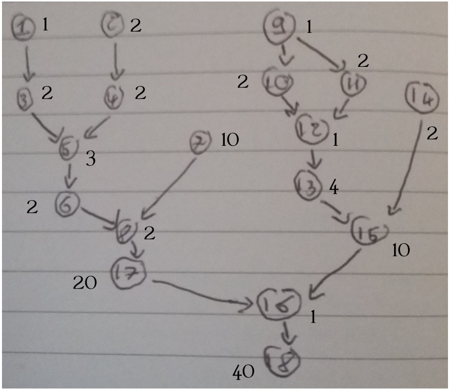
Here $\sum p_i = 107$

Width

Maximum number of tasks that do not have direct dependencies, or transitive dependencies.

Maximum number of independent tasks.

Sometimes called the longest antichain.

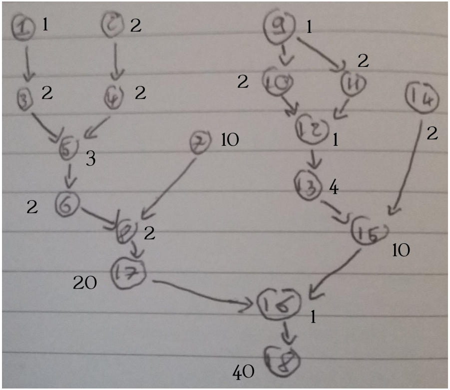


Width

Maximum number of tasks that do not have direct dependencies, or transitive dependencies.

Maximum number of independent tasks.

Sometimes called the longest antichain.



Here the width is 6.

Metrics: Width

Width

Maximum number of tasks that do not have direct dependencies, or transitive dependencies.

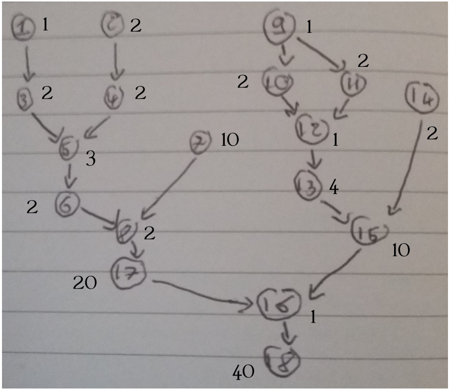
Maximum number of independent tasks.

Sometimes called the longest antichain.

Usage

Maximum number of useful processors.

$\forall P > Width, S(P) = S(Width)$



Here the width is 6.

Metrics: Width

Width

Maximum number of tasks that do not have direct dependencies, or transitive dependencies.

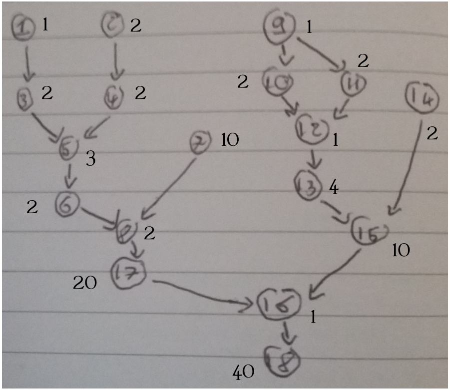
Maximum number of independent tasks.

Sometimes called the longest antichain.

Usage

Maximum number of useful processors.

$\forall P > Width, S(P) = S(Width)$



Here the width is 6.

How to find ?

Metrics: Width

Width

Maximum number of tasks that do not have direct dependencies, or transitive dependencies.

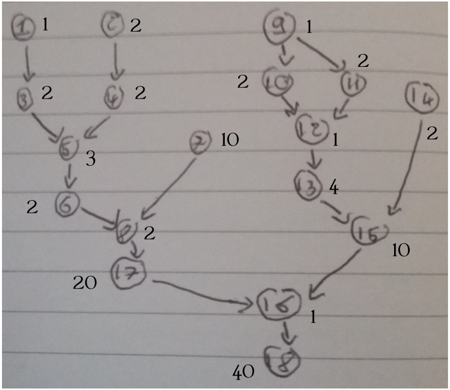
Maximum number of independent tasks.

Sometimes called the longest antichain.

Usage

Maximum number of useful processors.

$\forall P > Width, S(P) = S(Width)$



Here the width is 6.

How to find it ?

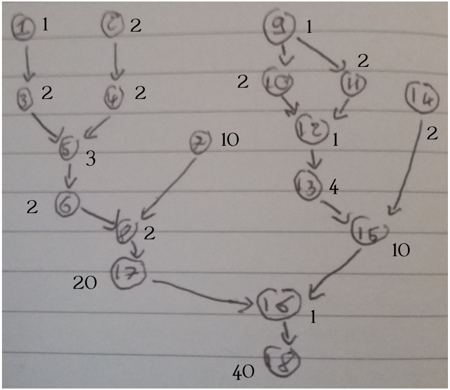
The problem is NP-Complete.
So trial and error.

Metrics: Critical Path

Critical Path

Longest chain of dependency (in term of processing time)

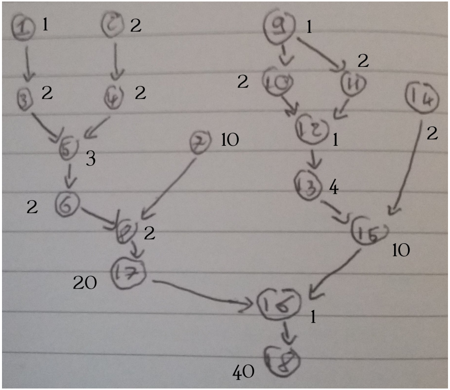
The length of the chain is often denoted CP, or T_∞.



Critical Path

Longest chain of dependency (in term of processing time)

The length of the chain is often denoted CP, or T_∞.



Here $7 \rightarrow 8 \rightarrow 17 \rightarrow 16 \rightarrow 18$.
$CP = 73$

Metrics: Critical Path

Critical Path

Longest chain of dependency (in term of processing time)

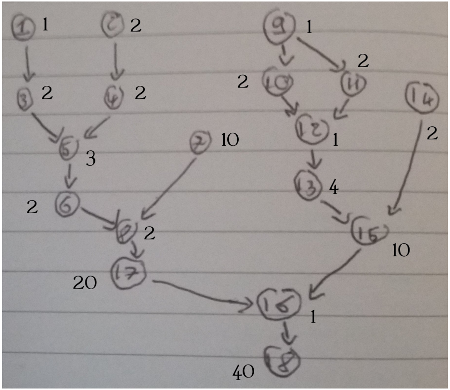
The length of the chain is often denoted CP, or T_∞.

Usage

Whichever way the algorithm unfolds, the critical path will have to be done.

The length of the critical path is a lower bound to the makespan

$C_{max} \geq CP$



Here $7 \rightarrow 8 \rightarrow 17 \rightarrow 16 \rightarrow 18$.
$CP = 73$

Metrics: Critical Path

Critical Path

Longest chain of dependency (in term of processing time)

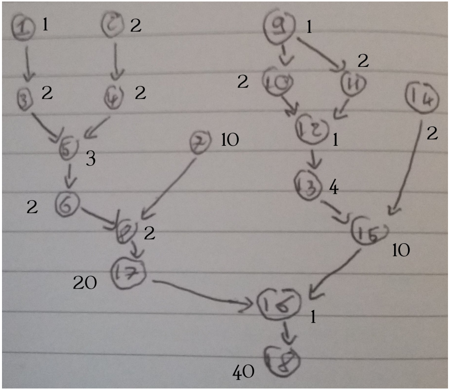
The length of the chain is often denoted CP, or T_∞.

Usage

Whichever way the algorithm unfolds, the critical path will have to be done.

The length of the critical path is a lower bound to the makespan

$C_{max} \geq CP$



Here $7 \rightarrow 8 \rightarrow 17 \rightarrow 16 \rightarrow 18$.
$CP = 73$

How to find it?

Metrics: Critical Path

Critical Path

Longest chain of dependency (in term of processing time)

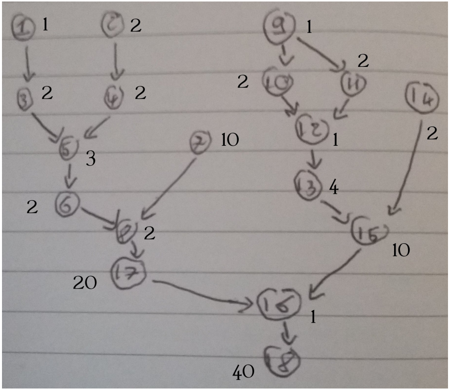
The length of the chain is often denoted CP, or T_∞.

Usage

Whichever way the algorithm unfolds, the critical path will have to be done.

The length of the critical path is a lower bound to the makespan

$C_{max} \geq CP$



Here $7 \rightarrow 8 \rightarrow 17 \rightarrow 16 \rightarrow 18$.

$CP = 73$

How to find it?

Recursively, from the roots down

$L(x) = p_x + \max_{y \in \Gamma^-(x)} L(y)$

Computing the critical path

An algorithm

1. start by marking the *level* all the sources of the DAG (the vertices without predecessors) with their processing times.
2. pick a vertex v where all predecessors are marked.
3. identify the predecessor of that vertex with the highest *level*.
4. add the processing time of vertex v to the highest level of the predecessor.
5. mark vertex v with the sum computed as the *level*
6. go back to step 2.

Activities

(see handout)

Outline

Scheduling

Problem

- A DAG of tasks
- Processing times p_i
- A number of processors

A schedule

Two functions mapping

- task t to processor $\pi(t)$
- task t to a time interval $[\sigma(t); C(t)[$. $C(t) = \sigma(t) + p_t$
- no two tasks execute on the same processor simultaneously.

Goal

Minimize $C_{max} = \max C(i)$

Scheduling

Problem

- A DAG of tasks
- Processing times p_i
- A number of processors

A schedule

Two functions mapping

- task t to processor $\pi(t)$
- task t to a time interval $[\sigma(t); C(t)[$. $C(t) = \sigma(t) + p_t$
- no two tasks execute on the same processor simultaneously.

Goal

Minimize $C_{max} = \max C(i)$

In practice

Lots of variants:

- with preemption (interrupting tasks).
- with migration (moving a task to an other processor).
- with restricted processor allocation (some tasks can only go on some processors).
- with unknown p_i
- sometimes the dependencies are only known at runtime.

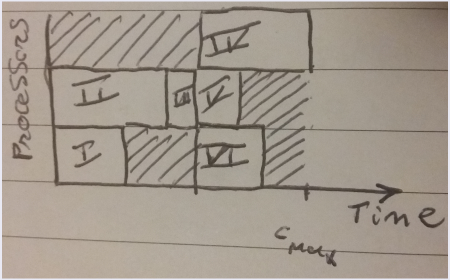
Representing schedules

Gantt Chart

A 2D depiction with time and processors as axes.

makes it easy to see the makespan

and idle times



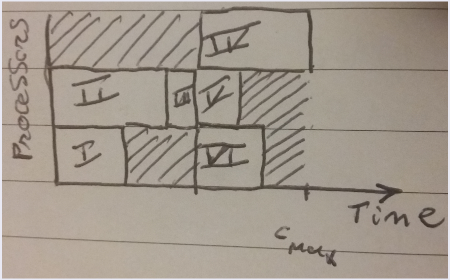
Representing schedules

Gantt Chart

A 2D depiction with time and processors as axes.

makes it easy to see the makespan

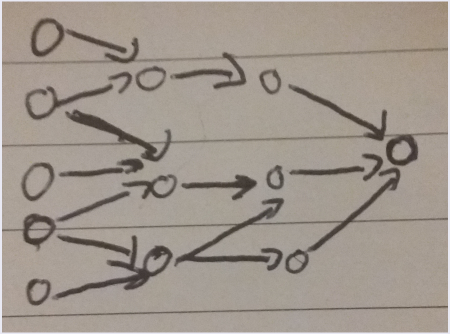
and idle times



Constrained DAG

Extract one chain per processor that respects dependency.

Highlights dependencies between processors.



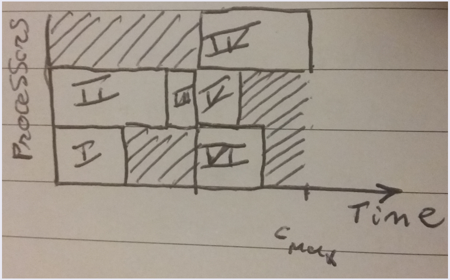
Representing schedules

Gantt Chart

A 2D depiction with time and processors as axes.

makes it easy to see the makespan

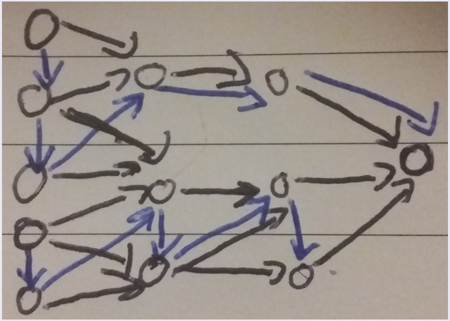
and idle times



Constrained DAG

Extract one chain per processor that respects dependency.

Highlights dependencies between processors.



Scheduling optimally is hard!

NP-Completeness

Finding the best schedule is an NP-hard problem.

(That means, you are unlikely to find the optimal in a short amount of time)

Scheduling optimally is hard!

NP-Completeness

Finding the best schedule is an NP-hard problem.

(That means, you are unlikely to find the optimal in a short amount of time)

Independent tasks variants

If all tasks are independent

still NP-Hard

If all tasks are independent and the number of processors is fixed

still NP-Hard (but weakly)

Scheduling optimally is hard!

NP-Completeness

Finding the best schedule is an NP-hard problem.

(That means, you are unlikely to find the optimal in a short amount of time)

Independent tasks variants

If all tasks are independent

still NP-Hard

If all tasks are independent and the number of processors is fixed

still NP-Hard (but weakly)

Variants

If all tasks have $p_i = 1$ (sometimes called UET)

still NP-Hard

If all tasks have $p_i = 1$, and makespan is 3

still NP-Hard

If all tasks have $p_i = 1$, and makespan is 3, and the graph is bipartite

still NP-Hard

Scheduling optimally is hard!

NP-Completeness

Finding the best schedule is an NP-hard problem.

(That means, you are unlikely to find the optimal in a short amount of time)

Independent tasks variants

If all tasks are independent

still NP-Hard

If all tasks are independent and the number of processors is fixed

still NP-Hard (but weakly)

Variants

If all tasks have $p_i = 1$ (sometimes called UET)

still NP-Hard

If all tasks have $p_i = 1$, and makespan is 3

still NP-Hard

If all tasks have $p_i = 1$, and makespan is 3, and the graph is bipartite

still NP-Hard

Scheduling is HARD!

List Scheduling for independent tasks is a 2-approximation algorithm

A greedy algorithm

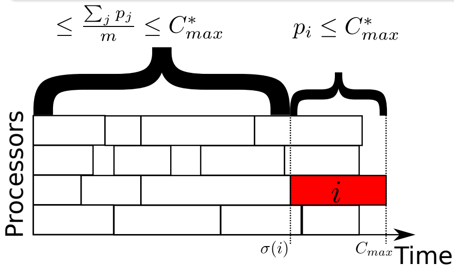
- Consider the tasks in any (unspecified) order.
- Pick a task and schedule it as early as possible on the first available processor.

List Scheduling for independent tasks is a 2-approximation algorithm

A greedy algorithm

Consider the tasks in any (unspecified) order.

Pick a task and schedule it as early as possible on the first available processor.



Proof

Let i be the last task to complete.

i starts at time $\sigma(i)$.

At time $\sigma(i)$, all the machines were busy.

$$\sigma(i) \leq \frac{\sum_j p_j}{m}$$

$$C_{max} = C(i) = \sigma(i) + p_i$$

$$C_{max} \leq \frac{\sum_j p_j}{m} + p_i$$

$$C_{max} \leq C^*_{max} + C^*_{max} \leq 2C^*_{max}$$

And if you do the equation well:

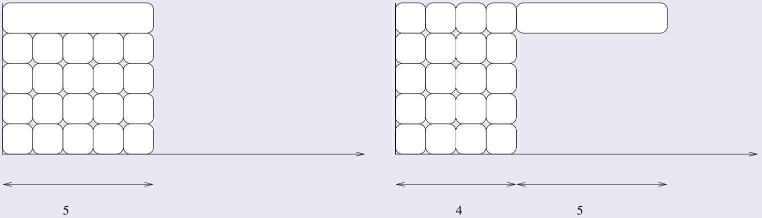
$$C_{max} \leq \frac{\sum_j p_j - p_{max}}{m} + p_{max}$$

How bad can List Scheduling really be ?

The idea

List Scheduling says "Given any order of the task". Let us use it to make it the worse possible.

Counter example



List Scheduling is a $(2 - \frac{1}{m})$-approximation algorithm and a counter example reaching this bound exists. The bound is said to be tight.

Is this the best algorithm ?

Is there a fast algorithm with guaranteed performance better than 2?

Largest Processing Time first

The idea

List Scheduling reaches 2 because the last task is the largest one.

Algorithm

Sort task by non-increasing order of processing times. Use List Scheduling.
Complexity $O(n \log n + nm)$

Approximation ratio of LPT

If the most loaded machine has one task : $C_{max} = p_{max}$ is optimal.

If the most loaded machine has two tasks : one can show the mapping is optimal (skipped)

If the most loaded machine has k tasks : the imbalance is less than $\frac{C_{max}}{k}$ and the ratio is $\frac{k+1}{k}$.

Theorem

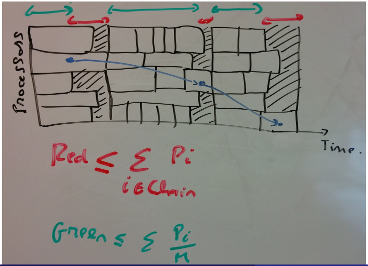
LPT is a $(\frac{4}{3} - \frac{1}{3m})$-approximation algorithm and the bound is tight.

List Scheduling for DAG

Algorithm

For DAGs, list scheduling is similar to the one for independent tasks.

> Pick the earliest available task.
>
> Schedule it on a free processor as early as possible.



LS is a 2-approximation

The makespan is decomposed in red and green part.
Red is one chain of the graph and is smaller than the longest chain in the graph (critical path).
Green is a fully occupied machine and is smaller the total work of the problem.
Both are smaller than a lowerbound on the best makespan.

$C_{max} \leq \frac{\sum p_i}{m} + \sum_{i \in CP} p_i$

$C_{max} \leq \frac{T_1}{m} + T_\infty$

$C_{max} \leq 2C^*_{max}$

By working the equation a bit harder: $C_{max} \leq \frac{T_1 - T_\infty}{m} + T_\infty$

Scheduling: Summary

Scheduling

- Finding for each task a processor for it to run on.
- Finding for each task a time interval for it to run on.

Finding the best schedule is hard! (even in sub-cases.)

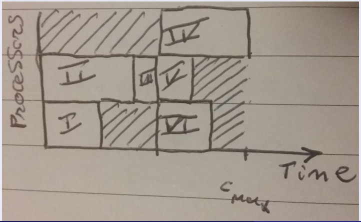
Scheduling: Summary

Scheduling

- Finding for each task a processor for it to run on.
- Finding for each task a time interval for it to run on.

Finding the best schedule is hard!
(even in sub-cases.)

Gantt Chart



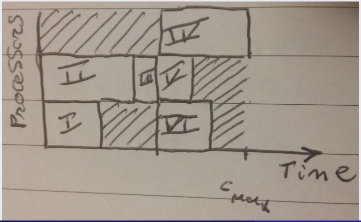
Scheduling: Summary

Scheduling

- Finding for each task a processor for it to run on.
- Finding for each task a time interval for it to run on.

Finding the best schedule is hard! (even in sub-cases.)

Gantt Chart



Three heuristic algorithms

For independent tasks:

- List Scheduling
  - Pick any available task
  - Schedule it ASAP
- LPT
  - Sort tasks by decreasing processing time
  - Then List Scheduling

For DAGs:

- List Scheduling
  - Pick the earliest available task
  - Schedule it ASAP

Activities

(see handout)

can you find the optimal schedule ?

Outline

Generalities

More art than science!

Answering questions like :

- Does it matter if these two things are swapped?
- Would the code still be correct if... ?
- Can we do something completely different?
- Is there a different expression that can compute the same value?

Analyze dependencies in sequential code

Code analysis

Granularize code

- statement
- loop iteration
- function call

The sequential code induces strict ordering. "Do these two tasks have to be this strictly ordered?"

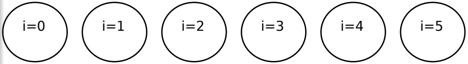
Weird Fibonacci

```
int fibo_v[N];

void fibo() {
  for (int i=0; i<6; ++i) {
    fibo_v[i] = 1;
  }
  for(int i=6; i<N; ++i) {
    fibo_v[i] = 0;
    for (int j=0; j<3; ++j) {
      fibo_v[i] += fibo_v[i-j-3];
    }
  }
}
```

Analyze dependencies in sequential code - example

Weird Fibonacci

```
int fibo_v[N];

void fibo() {
  for (int i=0; i<6; ++i) {
    fibo_v[i] = 1;
  }
  for(int i=6; i<N; ++i) {
    fibo_v[i] = 0;
    for (int j=0; j<3; ++j) {
      fibo_v[i] += fibo_v[i-j-3];
    }
  }
}
```



Let's start with the first loop.

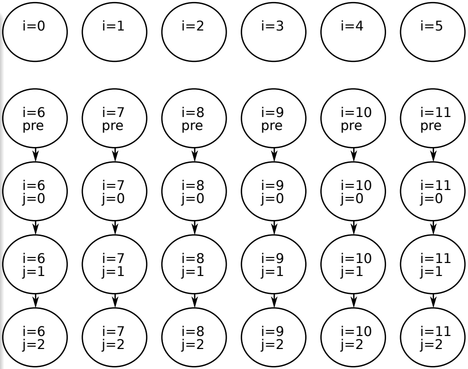
Make one vertex per loop iteration.

There are no dependencies.

Analyze dependencies in sequential code - example

Weird Fibonacci

```
int fibo_v[N];

void fibo() {
  for (int i=0; i<6; ++i) {
    fibo_v[i] = 1;
  }
  for(int i=6; i<N; ++i) {
    fibo_v[i] = 0;
    for (int j=0; j<3; ++j) {
      fibo_v[i] += fibo_v[i-j-3];
    }
  }
}
```

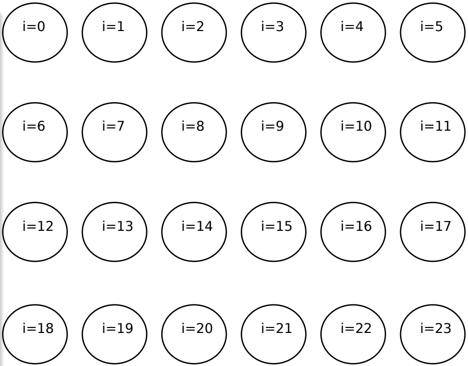


- Let's use the two nested loops
- Each iteration of j depends on the previous iteration
- Maybe some dependencies between different i iteration
- Let's forget the nested loop j and coarsen the graph (higher level)

Analyze dependencies in sequential code - example

Weird Fibonacci

```
int fibo_v[N];

void fibo() {
  for (int i=0; i<6; ++i) {
    fibo_v[i] = 1;
  }
  for(int i=6; i<N; ++i) {
    fibo_v[i] = 0;
    for (int j=0; j<3; ++j) {
      fibo_v[i] += fibo_v[i-j-3];
    }
  }
}
```



Let's look at some iteration of i, say until 23.

These are just the tasks

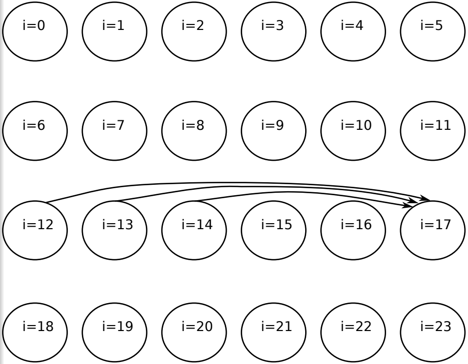
What are the dependencies ?

Let's consider just task $i = 17$ for the moment

Analyze dependencies in sequential code - example

Weird Fibonacci

```
int fibo_v[N];

void fibo() {
  for (int i=0; i<6; ++i) {
    fibo_v[i] = 1;
  }
  for(int i=6; i<N; ++i) {
    fibo_v[i] = 0;
    for (int j=0; j<3; ++j) {
      fibo_v[i] += fibo_v[i-j-3];
    }
  }
}
```



$i = 17$ will read *fibo_v*[12], *fibo_v*[13], *fibo_v*[14]

$i = 12$ writes *fibo_v*[12]

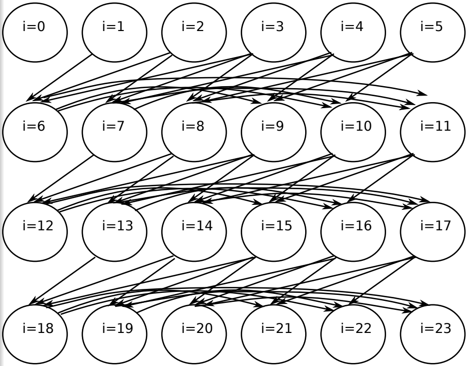
So $i = 17$ can not happen before task $i = 12$ completes

Similarly, $i = 17$ depends on $i = 13$ and $i = 14$ completions

Analyze dependencies in sequential code - example

Weird Fibonacci

```
int fibo_v[N];

void fibo() {
  for (int i=0; i<6; ++i) {
    fibo_v[i] = 1;
  }
  for(int i=6; i<N; ++i) {
    fibo_v[i] = 0;
    for (int j=0; j<3; ++j) {
      fibo_v[i] += fibo_v[i-j-3];
    }
  }
}
```



Similarly, all tasks for $i \geq 6$, have 3 inputs.

Can you find the width, work, CP of this graph?

As a function of N?

Usual dependencies

$x \rightarrow y$

When x is before y in sequential and there is

flow dependence (read-after-write (RAW))

y reads a variable writen by x

$out[x] = 12; in[y] += out[x];$

anti-dependence (write-after-read (WAR))

y writes a variable read by x

$out[x] += in[y]; in[y] += 12;$

output-dependence (write-after-write (WAW))

y writes a variable writen by x

$glob = x; glob = y$

Note that input-dependence (read-after-read (RAR)) usually does not matter.

How to extract dependencies? A recipe (1)

Granularity

Choose what will be a task: an uninterupted sequence of instructions

- usually, an iteration of a loop.
  - different values of i for a single loop algorithm
  - different pairs (i, j) for a 2 loop algorithm
  - need to introduce a *pre* and *post* tasks for each iteration
- or a particular call to a function
  - for recursive algorithms
  - one task per MergeSort(i,j)

Assign each task with a processing time.

How to extract dependencies? A recipe (2)

List variable access

For each task, identify which variable is accessed

Decide whether the access if Read, Write, or ReadWrite

If branching happens that can not decided without knowing the data, assume both branch can happen.

(If the branch depends on the task id, it can be known)

Find dependencies

If two tasks x and y access the same variable

And one of the access is a Write access

Add a dependency from the earlier task to the later task

Example

Matvec on the board

Par Graph Lib

https://github.com/esaule/par_graph_lib

This is a library written to help you with understanding how this works.

The library enables creating tasks inside an exisiting code

And declaring variable access.

It does not extract the dependencies for you. You have to decide them.

It does not support early terminating algorithms.

Uses Bridges for visualization.

Mutual exclusion and resolution

Mutual exclusion

Appropriate when two blocks of code can not run simultaneously.

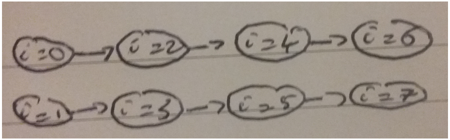
But either order is valid.

Essentially pairs a conflict graph to the DAG.

Note on the graph to express mutual exclusion.

Example

```c
int val[N];
int re[2];

int f (int i);

void red () {
  for (int i=0; i<N; ++i) {
    val[i] = f(i);
    re[i%2] += val[i];
  }
}
```



Mutual exclusion and resolution

Mutual exclusion

Appropriate when two blocks of code can not run simultaneously.

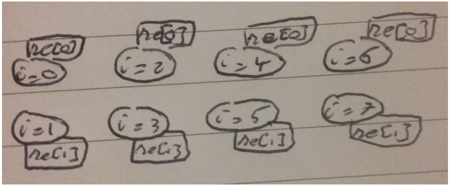
But either order is valid.

Essentially pairs a conflict graph to the DAG.

Note on the graph to express mutual exclusion.

Example

```c
int val[N];
int re[2];

int f (int i);

void red () {
  for (int i=0; i<N; ++i) {
    val[i] = f(i);
    re[i%2] += val[i];
  }
}
```



Mutual exclusion and resolution

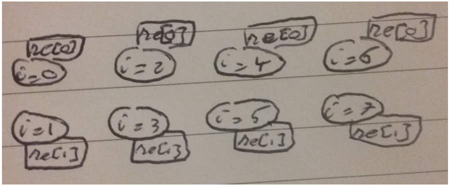
Mutual exclusion

- Appropriate when two blocks of code can not run simultaneously.
- But either order is valid.
- Essentially pairs a conflict graph to the DAG.
- Note on the graph to express mutual exclusion.

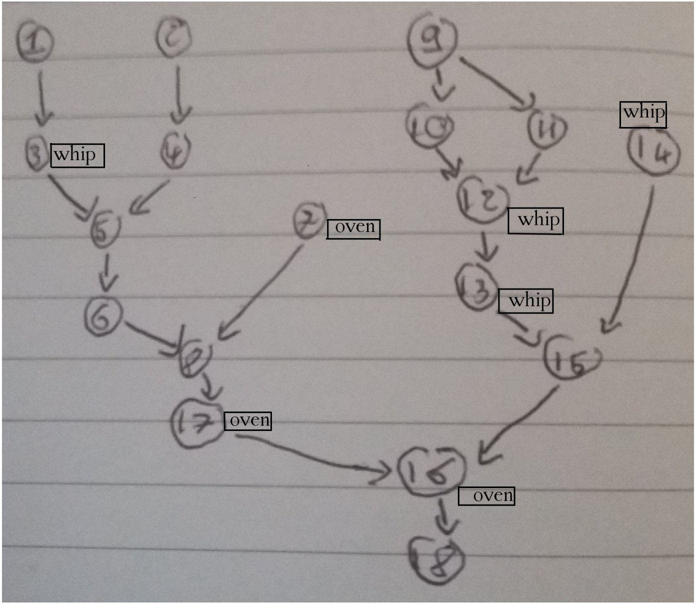
Not too much help!

Example

```
int val[N];
int re[2];

int f (int i);

void red () {
  for (int i=0; i<N; ++i) {
    val[i] = f(i);
    re[i%2] += val[i];
  }
}
```



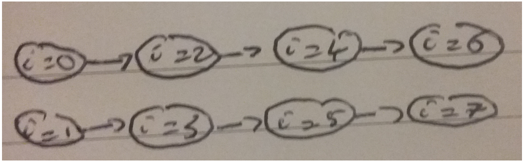
A story of whip and oven



And now for something completely different

Example

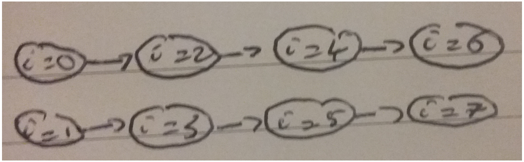
```c
void red () {
  for (int i=0; i<N; ++i) {
    val[i] = f(i);
    re[i%2] += val[i];
  }
}
```



And now for something completely different

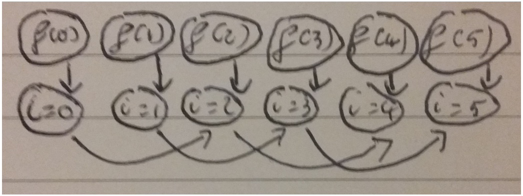
Example

```
void red () {
  for (int i=0; i<N; ++i) {
    val[i] = f(i);
    re[i%2] += val[i];
  }
}
```



Do it differently

```
void red () {
  for (int i=0; i<N; ++i)
    val[i] = f(i);
  for (int i=0; i<N; ++i)
    re[i%2] += val[i];
}
```



Changes dependency structures.

Useful if `f(i)` is expensive.

Assumes `f(i)`s are independent.

Outline

Assignment

(see text.)

Extract parallelism

- Transform
- Reduction (int +, string +, float +)
- Find First (array and list)
- Prefix Sum
- Merge Sort

Outline

External

cilk on graphs metrics

Conflict graph and coloring:

- Conflict graphs: `http://math.cmu.edu/~bkell/21110-2010s/conflict-graphs.html`
- A. H Gebremedhin, F. Manne, Alex Pothen. What Color Is Your Jacobian? Graph Coloring for Computing Derivatives. Siam Review 2005.
- M. Deveci, E. Boman, K. Devine, and S. Rajamanickam. Parallel Graph Coloring for Manycore Architectures. IPDPS 2016.

Scheduling:

- Scheduling is NP-Hard: M. Garey and D. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman. 1979.
- LS for independent tasks: R. Graham. Bounds for certain multiprocessing anomalies. Bell System Technical Journal. 1966
- LPT and LS with precedence: R. Graham. Bounds on Multiprocessing Timing Anomalies. SIAM Journal on Applied Mathematics. 1969.
- Chapter 1 and 7 of. H. Casanova, A. Legrand, Y. Robert. Parallel Algorithms, CRC Press. 2008

Dependency extraction:

- A. J. Bernstein. Analysis of programs for parallel processing. IEEE Transactions on Electronic Computers, 15:757–762, Oct. 1966.

Software:

- Cilk Plus extract dependencies with the programmers help: `https://software.intel.com/en-us/node/522598`
- Athapascan/KAAPI does something similar: `https://hal.inria.fr/inria-00069901/document`

Typical compiler optimization:

- Loop fission: `https://en.wikipedia.org/wiki/Loop_fission`
- Loop tiling: `https://en.wikipedia.org/wiki/Loop_tiling`
- Various: `https://en.wikipedia.org/wiki/Compiler_optimization`