Extracting Parallelism: From Code to Parallel Task Graph

Jinzhen Wang
jwang96@charlotte.edu

Department of Computer Science
UNC Charlotte

Learning Outcomes

Lecture

At the end of this session you will know how to

- Find the dependencies in a sequential code (or algorithm)
- Express the dependencies as a Parallel Task Graph

Serial Program $\rightarrow$ Parallel Program

- We discussed scaling the computation via *strong scaling* and *weak scaling*.
- The key now, becomes
  - Find the portion $(1-x)$ that can be parallelized $\rightarrow$ **Extract the Parallelism**
- But, it is not easy. Sometimes more art than science!
- In the design phase, you probably need to consider lots of things:
  - Does it matter if these two things are swapped Would the code still be correct if I ...?
  - Can we do something completely different?
  - Is there a different expression that can compute the same value?
  - ...

Code Example

- Calculate Fibonacci Number:

$$F0 = 0, F1 = 1,$$
$$Fn = F_{n-1} + F_{n-2} \text{ for } n > 1$$

- How do we usually implement one?
  - Recursively?
  - Loop-based?

Implementation: Recursive approach

```
int fibo (int p) {
  if (p < 2) {
    return 1;
  }
  int p1 = p-1;
  int p2 = p-2;

  int r1 = fibo(p1);
  int r2 = fibo(p2);

  return r1 + r2;
}
```

Implementation: Recursive approach

```
int fibo (int p) {
  if (p < 2) {
    return 1;
  }
  int p1 = p-1;
  int p2 = p-2;

  int r1 = fibo(p1);
  int r2 = fibo(p2);

  return r1 + r2;
}
```

Pause and Think

- Can it be parallelized?

Implementation: Recursive approach

```
int fibo (int p) {
  if (p < 2) {
    return 1;
  }
  int p1 = p-1;
  int p2 = p-2;

  int r1 = fibo(p1);
  int r2 = fibo(p2);

  return r1 + r2;
}
```

Pause and Think

- Can it be parallelized?
- It is very hard because each calculation F(x) depends on previous calculation of F(x-1).

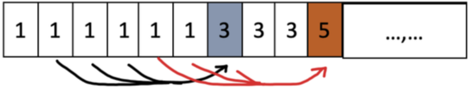
Another code example: Weird Fibonacci

Weird Fibonacci

$F_n = F_{n-3} + F_{n-4} + F_{n-5}$.

```
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];
    }
  }
}
```

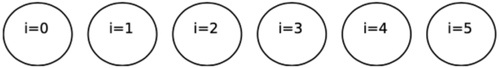


Any parallelism?

Analyze dependencies in sequential code

```
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];
    }
  }
}
```

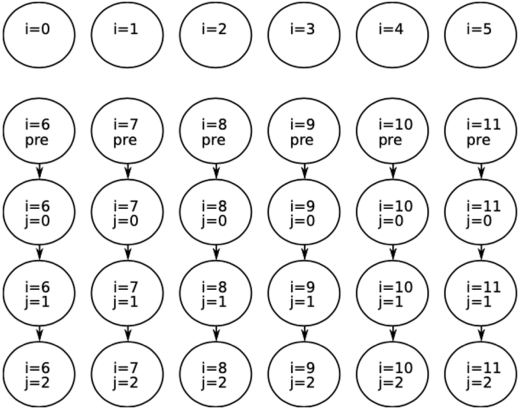


- Let's start with the first loop.
- Make one vertex per loop iteration.
- All vertices can be executed at the same time → No dependencies.

Analyze dependencies in sequential code

```
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];
    }
  }
}
```

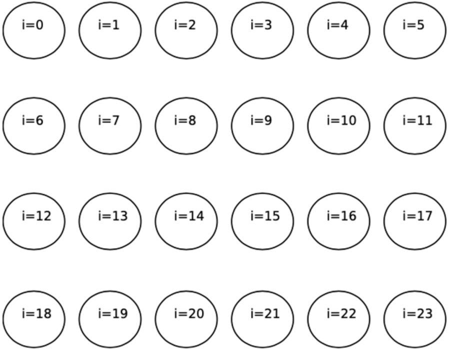


- Let's start with the second loop.
- Each iteration of j depends on the previous iteration.
- Maybe some dependencies between different i iteration

Analyze dependencies in sequential code

```
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];
    }
  }
}
```

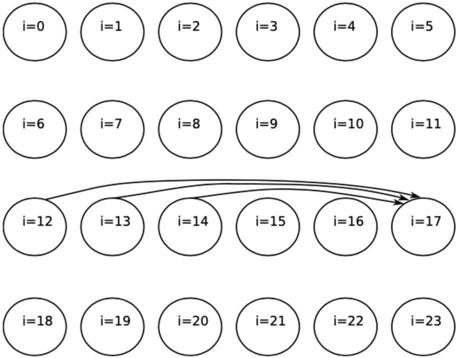


- Let's look at some iteration of *i*, say until 23.
- These are just the tasks
- What are the dependencies ?
- Let's consider just task i $= 17$ for the moment

Analyze dependencies in sequential code

```
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];
    }
  }
}
```

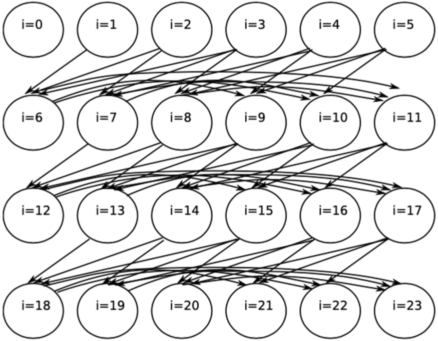


- $i = 17$ will read *fibo_v*[12], *fibo_v*[13], *fibo_v*[14]
- $i = 12$ writes *fibo_v*[12]
- So $i = 17$ can not happen before task $i = 12$ completes
- Similarly, $i = 17$ depends on $i = 13$ and $i = 14$ completions

Analyze dependencies in sequential code

```
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

Analyze dependencies in sequential code - example

- Work
  - Total amount of work that is to perform on the application executing the code, assume both branches can happen
  - (The time to complete the task, with a single process)
- Width (aka Parallelism)
  - How many threads can work at once.
- Critical Path
  - Length of the longest chain of task to execute.

Analyze dependencies in sequential code - example

- Width: 6 (The first 6 tasks)
- Work: 24 (Assuming all task have a cost of 1; not quite true but constant time)
- Critical Path: 5, 8, 11, 14, 17, 20, 23. Length : 7

Analyze dependencies in sequential code - example

For calculating until n

- Width: 6 (The first 6 tasks). $O(1)$
- Work: $\Theta(n)$
- Critical Path: roughly $n/3 = \Theta(n)$

When is there a dependency $x \to y$?

x has to be before y in the sequential execution:

Flow dependence (Read After Write)

y reads a variable writen by x.

x:
common += foo;

y:
bar += common;

Then $x \to y$

Output-dependence (Write-After-Write)

y writes a variable writen by x.

x:
common = 2 * foo;

y:
common = sqrt(bar);

Then $x \to y$

Anti-dependence (Write-After-Read)

y writes a variable read by x.

x:
foo += common;

y:
common = bar;

Then $x \to y$;

Input-dependence (Read-After-Read)

y reads a variable read by x.

x:
foo = 2 * common;

y:
bar = sqrt(common);

This does **not** create a dependency.

How to extract dependencies? A recipe (1)

Granularity

Choose what will be a task: an uninterrupted 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)

Note that all tasks must be known in advance. So you can not have a code with complex loop termination or break.

Assign each task with a processing time.

How to extract dependencies? A recipe (2)

List variable access

- For each task, identify which variables are accessed
- Decide whether the access is a Read, Write, or ReadWrite access
- If branching happens that can not decided without knowing the data, assume both branch can happen.
    - (If the branch depends uniquely 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

Cleanup

- Remove transitive dependencies

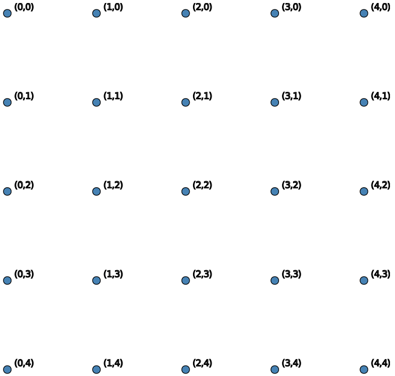
A 2D example

https://leetcode.com/problems/longest-common-subsequence/description/

Longest Common Subsequence

```
int LCSLength(char* X, int m,
              char* Y, int n,
       vector<vector<int>> C) {
  for (int i=0; i <= m; ++i)
    C[i][0] = 0;
  for (int j=1; j <= n; ++j)
    C[0][j] = 0;
  for (int a=1; a <= m; ++a)
    for (int b=1; b <= n; ++b)
      if (X[a-1] == Y[b-1])
        C[a][b] = C[a-1][b-1] + 1;
      else
        C[a][b] = max(C[a][b-1],
                      C[a-1][b]);
  return C[m][n];
}
```

Step 1: Decide on what is a task



Here I chose to use each iteration of each loop as a task.
Picture for $n = 4$, $m = 4$, but can be generalized.
All tasks have constant time complexity : $\Theta(1)$

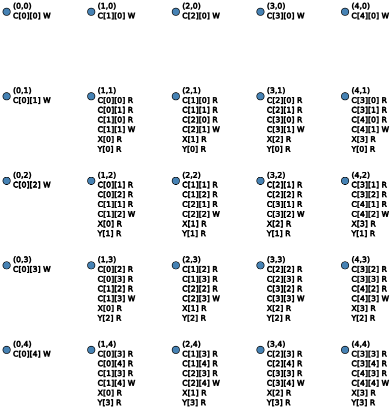
A 2D example

https://leetcode.com/problems/longest-common-subsequence/description/

Longest Common Subsequence

```
int LCSLength(char* X, int m,
              char* Y, int n,
       vector<vector<int>> C) {
  for (int i=0; i <= m; ++i)
    C[i][0] = 0;
  for (int j=1; j <= n; ++j)
    C[0][j] = 0;
  for (int a=1; a <= m; ++a)
    for (int b=1; b <= n; ++b)
      if (X[a-1] == Y[b-1])
        C[a][b] = C[a-1][b-1] + 1;
      else
        C[a][b] = max(C[a][b-1],
                      C[a-1][b]);
  return C[m][n];
}
```

Step 2: Figure out variable access

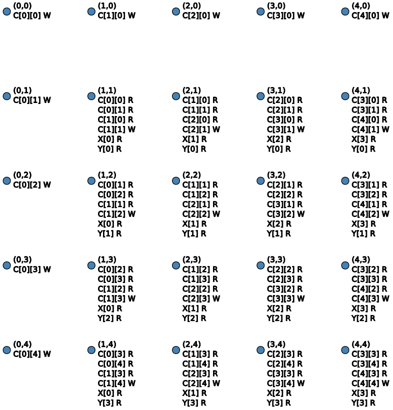


Need to have three reads on C even if not all three will happen. In case of an if statement that depends on data, all possibilities must be accounted for.

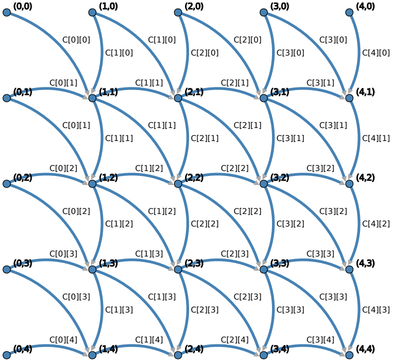
A 2D example

`https://leetcode.com/problems/longest-common-subsequence/description/`

Step 2: Figure out variable access



Step 3: Identify dependencies

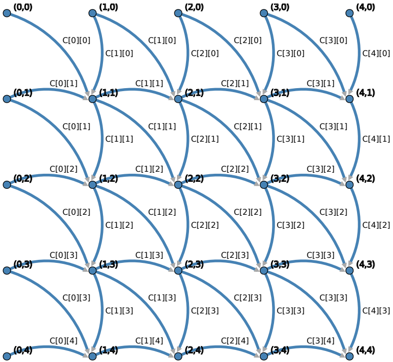


All dependencies come from accesses to `C`.

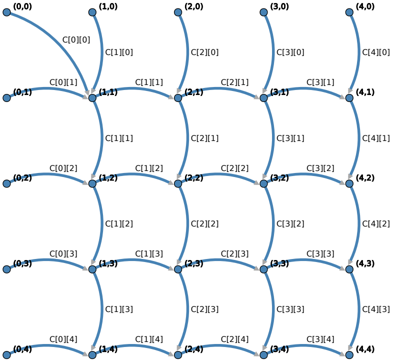
A 2D example

https://leetcode.com/problems/longest-common-subsequence/description/

Step 3: Identify dependencies



Step 4: Remove transitive dependencies



Most diagonal dependencies are implied by the other ones.

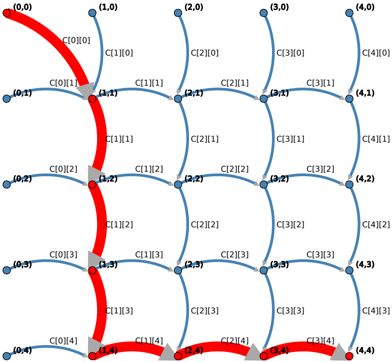
A 2D example

https://leetcode.com/problems/longest-common-subsequence/description/

Longest Common Subsequence

```
int LCSLength(char* X, int m,
              char* Y, int n,
       vector<vector<int>> C) {
  for (int i=0; i <= m; ++i)
    C[i][0] = 0;
  for (int j=1; j <= n; ++j)
    C[0][j] = 0;
  for (int a=1; a <= m; ++a)
    for (int b=1; b <= n; ++b)
      if (X[a-1] == Y[b-1])
        C[a][b] = C[a-1][b-1] + 1;
      else
        C[a][b] = max(C[a][b-1],
                      C[a-1][b]);
  return C[m][n];
}
```

Step 5: Compute Metrics

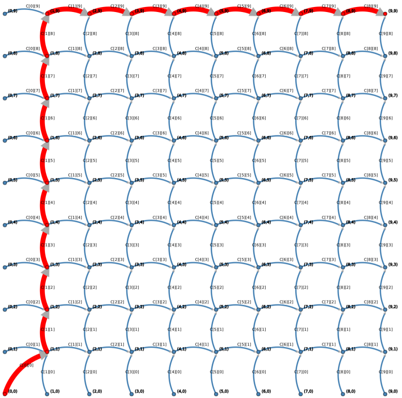


Work: 25 (25 task, all in O(1))
Width: 9 (first row and first column)
Length Critical Path: 8

The parallelism of Longest Common Subsequence

Longest Common Subsequence

```
int LCSLength(char* X, int m,
              char* Y, int n,
       vector<vector<int>> C) {
  for (int i=0; i <= m; ++i)
    C[i][0] = 0;
  for (int j=1; j <= n; ++j)
    C[0][j] = 0;
  for (int a=1; a <= m; ++a)
    for (int b=1; b <= n; ++b)
      if (X[a-1] == Y[b-1])
        C[a][b] = C[a-1][b-1] + 1;
      else
        C[a][b] = max(C[a][b-1],
                      C[a-1][b]);
  return C[m][n];
}
```



Width: $\Theta(n+m)$

Work: $\Theta(nm)$

Critical Path: $\Theta(n+m)$

What About Recursive Codes?

Recursive Fibonacci

```c
int fibo (int p) {
  if (p < 2) {
    return 1;
  }
  int p1 = p-1;
  int p2 = p-2;

  int r1 = fibo(p1);
  int r2 = fibo(p2);

  return r1 + r2;
}
```

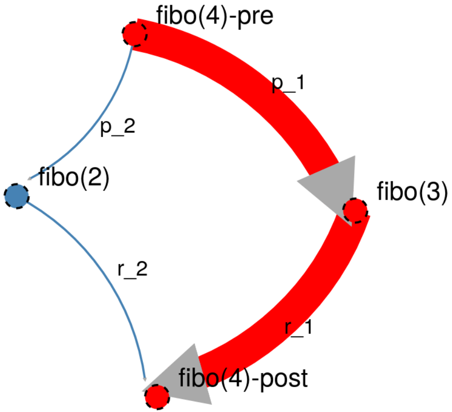
What About Recursive Codes?

Recursive Fibonacci

```
int fibo (int p) {
  if (p < 2) {
    return 1;
  }
  int p1 = p-1;
  int p2 = p-2;

  int r1 = fibo(p1);
  int r2 = fibo(p2);

  return r1 + r2;
}
```



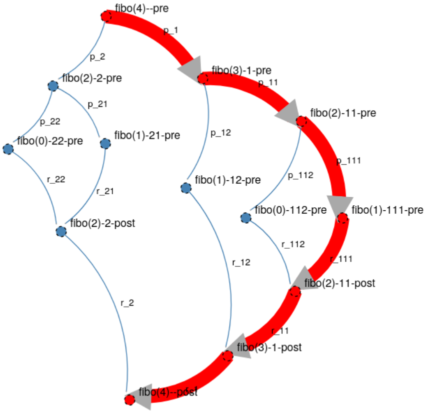
What About Recursive Codes?

Recursive Fibonacci

```
int fibo (int p) {
  if (p < 2) {
    return 1;
  }
  int p1 = p-1;
  int p2 = p-2;

  int r1 = fibo(p1);
  int r2 = fibo(p2);

  return r1 + r2;
}
```



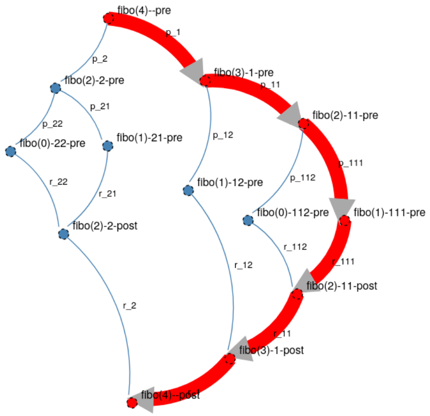
What About Recursive Codes?

Recursive Fibonacci

```
int fibo (int p) {
  if (p < 2) {
    return 1;
  }
  int p1 = p-1;
  int p2 = p-2;

  int r1 = fibo(p1);
  int r2 = fibo(p2);

  return r1 + r2;
}
```



Work $= \Theta(fibo(n))$
CP $= \Theta(n)$
Width $= \Theta(fibo(n))$

External

Dependency extraction:

- The origins of the model: A. J. Bernstein. Analysis of programs for parallel processing. IEEE Transactions on Electronic Computers, 15:757–762, Oct. 1966.
- Voevodin V., Antonov A., Voevodin V. (2018) What Do We Need to Know About Parallel Algorithms and Their Efficient Implementation?. In: Prasad S., Gupta A., Rosenberg A., Sussman A., Weems C. (eds) Topics in Parallel and Distributed Computing.
- Chapter 2 to 5.1 of Oliver Sinnen. Task Scheduling for Parallel Systems. John Wiley & Sons, Inc. 2007. Access it through the library: https://librarylink.uncc.edu/login?url=https://onlinelibrary.wiley.com/doi/book/10.1002/0470121173
- Chapter 1 and 7 of. H. Casanova, A. Legrand, Y. Robert. Parallel Algorithms, CRC Press. 2008

Software:

- Par graph lib: https://github.com/esaule/par_graph_lib
- 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