# CS3383 Unit 4: dynamic multithreaded algorithms

David Bremner

March 20, 2024

# Outline

Dynamic Multithreaded Algorithms

- Fork-Join Model
- Span, Work, And Parallelism
- Parallel Loops

# Introduction to Parallel Algorithms

## Dynamic Multithreading

- Also known as the *fork-join* model
- Shared memory, *multicore*
- Cormen et. al 4th edition, Chapter 26

# Introduction to Parallel Algorithms

## Dynamic Multithreading

- Also known as the *fork-join* model
- Shared memory, *multicore*
- Cormen et. al 4th edition, Chapter 26

## Nested Parallelism

- Spawn a subroutine, carry on with other work.
- Similar to `fork` in POSIX.

# Introduction to Parallel Algorithms

## Nested Parallelism

- Spawn a subroutine, carry on with other work.
- Similar to `fork` in POSIX.

## Parallel Loop

- iterations of a for loop *can* execute in parallel.
- Like `OpenMP` parallel for, Python `multiprocessing` parallel map.

# Writing parallel (pseudo)-code

## Keywords

| | |
|---|---|
| parallel | for loop iterations are (potentially) concurrent |
| spawn | Run the procedure (potentially) concurrently |
| sync | Wait for all spawned children to complete. |

# Writing parallel (pseudo)-code

## Keywords

| | |
|---|---|
| parallel | for loop iterations are (potentially) concurrent |
| spawn | Run the procedure (potentially) concurrently |
| sync | Wait for all spawned children to complete. |

## Serialization

- remove keywords from parallel code yields correct serial code
- Adding parallel keywords to correct serial code might break it (e.g. race conditions).

# Fibonacci Example

**function** $\textsc{Fib}(n)$
    **if** $n \leq 1$ **then**
        return $n$
    **else**
        $x = \text{Fib}(n-1)$
        $y = \text{Fib}(n-2)$

        return $x + y$
    **end if**
**end function**

# Fibonacci Example

```
function FIB(n)
    if n ≤ 1 then
        return n
    else
        x = spawn Fib(n − 1)
        y = Fib(n − 2)
        sync
        return x + y
    end if
end function
```

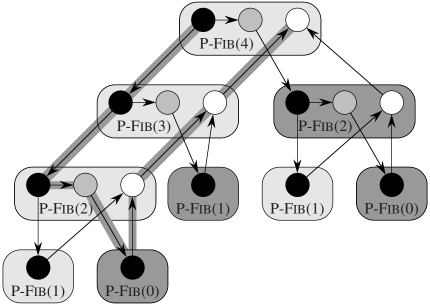
# Fibonacci example in OpenMP

```
long fib(int n) {
  long x, y;
  if (n<=1)
    return n;
  else {
    #pragma omp task shared(x)
    x=fib(n-1);
    y=fib(n-2);
    #pragma omp taskwait
    return x+y;
  }
}
```

# Computation DAG

Strands: Sequential instructions with no *parallel*, *spawn*, return from *spawn*, or *sync*.

**function** $\textsc{Fib}(n)$
  **if** $n \leq 1$ **then** $\triangleright$ ●
    return $n$
  **else**
    $x =$ spawn $\text{Fib}(n-1)$
    $y = \text{Fib}(n-2)$ $\triangleright$ ●
    sync
    return $x + y$ $\triangleright$ ○
  **end if**
**end function**

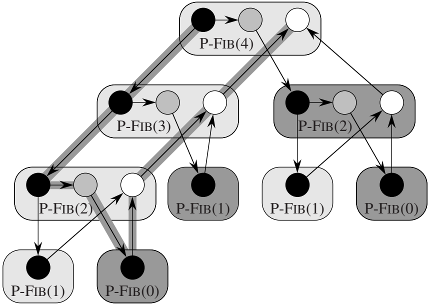


# Computation DAG

Strands: Sequential instructions with no *parallel*, *spawn*, return from *spawn*, or *sync*.

nodes strands

down edges spawn



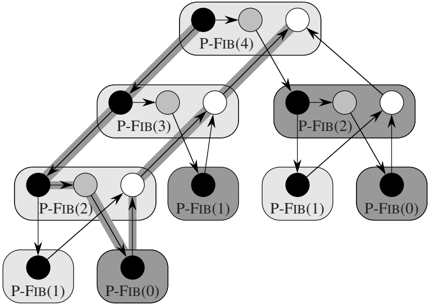
# Computation DAG

Strands: Sequential instructions with no *parallel*, *spawn*, return from *spawn*, or *sync*.

nodes strands

down edges spawn

up edges return



# Computation DAG

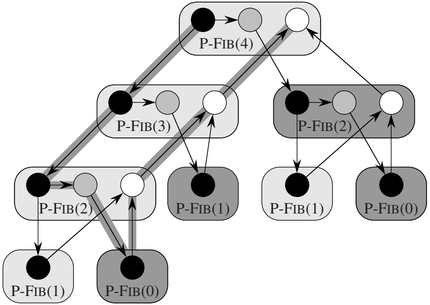
Strands: Sequential instructions with no *parallel*, *spawn*, return from *spawn*, or *sync*.

nodes strands

down edges spawn

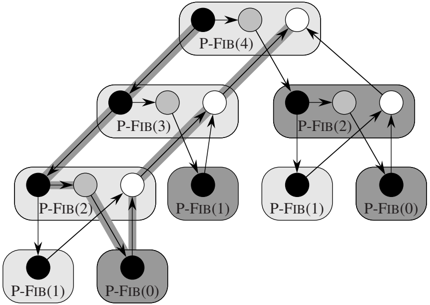
up edges return

horizontal edges sequential



# Computation DAG

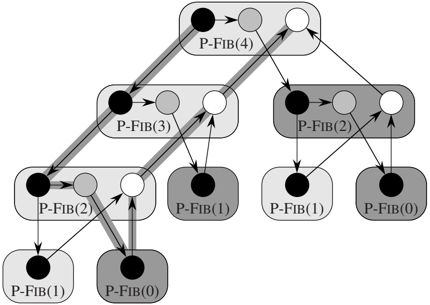
Strands: Sequential instructions with no *parallel*, *spawn*, return from *spawn*, or *sync*.




| | |
|---|---|
| nodes | strands |
| down edges | spawn |
| up edges | return |
| horizontal edges | sequential |
| critical path | longest path in DAG |

# Computation DAG

Strands: Sequential instructions with no *parallel*, *spawn*, return from *spawn*, or *sync*.




| | |
|---|---|
| nodes | strands |
| down edges | spawn |
| up edges | return |
| horizontal edges | sequential |
| critical path | longest path in DAG |
| span | weighted length of critical path ≡ lower bound on time |

# Work and Speedup

$T_1$ *Work*, sequential time.

# Work and Speedup

$T_1$ *Work*, sequential time.

$T_p$ Time on $p$ processors.

# Work and Speedup

$T_1$ *Work*, sequential time.

$T_p$ Time on $p$ processors.

## Work Law

$$T_p \geq T_1/p$$

$$\text{speedup} := T_1/T_p \leq p$$

# Parallelism

$T_p$ Time on $p$ processors.

# Parallelism

We could idle processors:

$$T_p \geq T_\infty \tag{1}$$

$T_p$ Time on $p$ processors.

$T_\infty$ *Span*, time given unlimited processors.

# Parallelism

We could idle processors:

(1) $$T_p \geq T_\infty$$

Best possible speedup:

$$\text{parallelism} = T_1/T_\infty \geq T_1/T_p = \text{speedup}$$

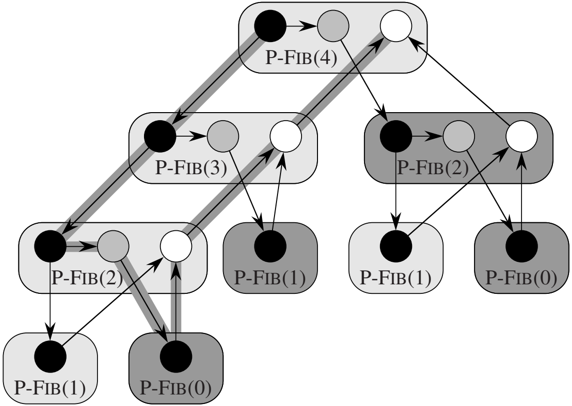
$T_p$ Time on $p$ processors.

$T_\infty$ *Span*, time given unlimited processors.

# Span and Parallelism Example

Assume strands are unit cost.

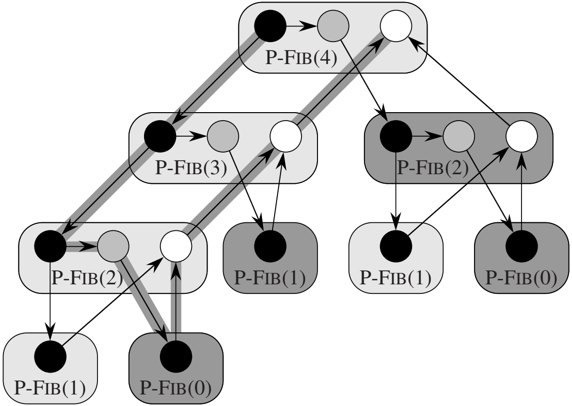
- $T_1 = 17$



# Span and Parallelism Example

Assume strands are unit cost.

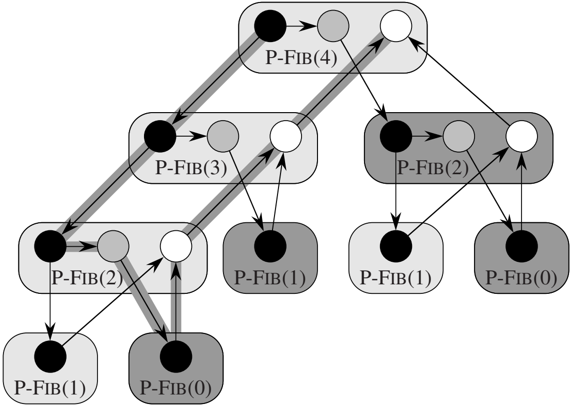
- $T_1 = 17$
- $T_\infty = 8$



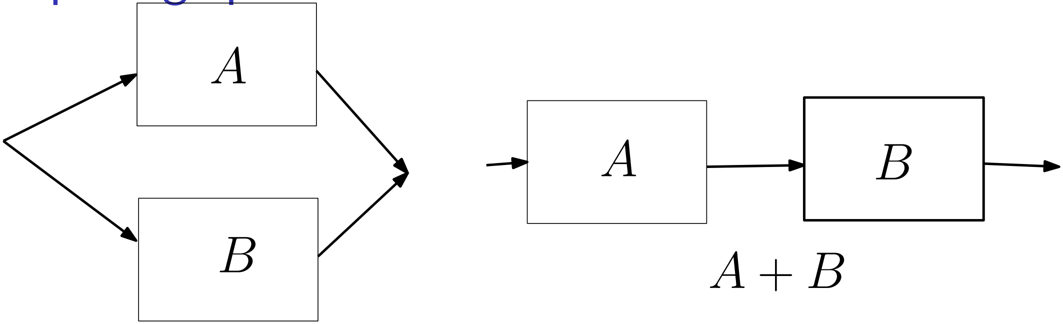
# Span and Parallelism Example

Assume strands are unit cost.

- $T_1 = 17$
- $T_\infty = 8$
- Parallelism $= 2.125$ for this input size.



# Composing span and work

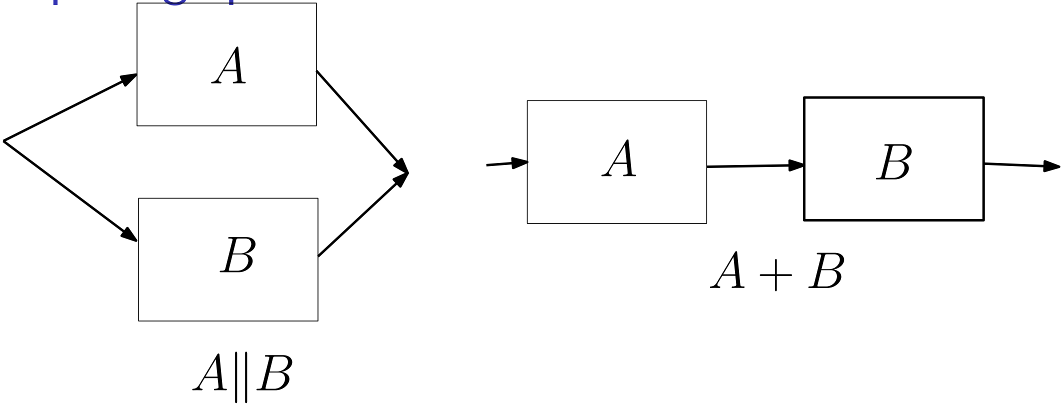



$A \| B$

$A + B$

series $T_\infty(A + B) = T_\infty(A) + T_\infty(B)$

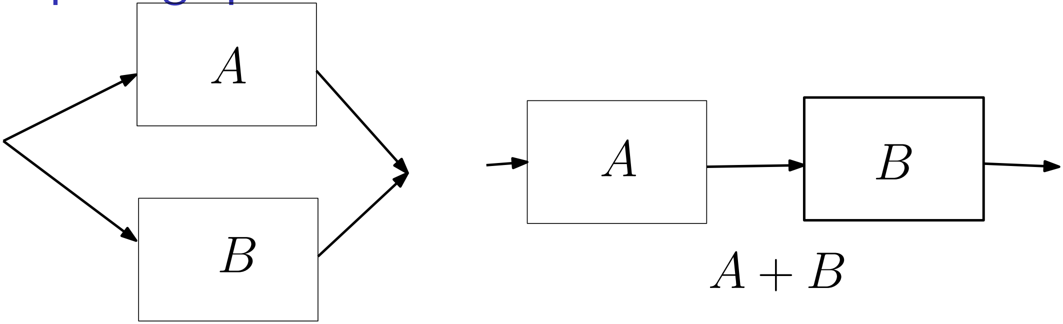
# Composing span and work




series $T_\infty(A + B) = T_\infty(A) + T_\infty(B)$

parallel $T_\infty(A \| B) = \max(T_\infty(A), T_\infty(B))$

# Composing span and work




$A \| B$

$A + B$

series $T_\infty(A + B) = T_\infty(A) + T_\infty(B)$

parallel $T_\infty(A \| B) = \max(T_\infty(A), T_\infty(B))$

series or parallel $T_1 = T_1(A) + T_1(B)$

# Work of Parallel Fibonacci I/II

Write $T(n)$ for $T_1$ on input $n$.

$$T(n) = T(n-1) + T(n-2) + \Theta(1)$$

# Work of Parallel Fibonacci I/II

Write $T(n)$ for $T_1$ on input $n$.

$$T(n) = T(n-1) + T(n-2) + \Theta(1)$$

Let $\phi \approx 1.62$ be the solution to

$$\phi^2 = \phi + 1$$

# Work of Parallel Fibonacci I/II

Write $T(n)$ for $T_1$ on input $n$.

$$T(n) = T(n-1) + T(n-2) + \Theta(1)$$

Let $\phi \approx 1.62$ be the solution to

$$\phi^2 = \phi + 1$$

We can show by induction (twice) that

$$T(n) \in \Theta(\phi^n)$$

# Work of Parallel Fibonacci II/II

(I.H.) $T(n) \leq a\phi^n - b$

# Work of Parallel Fibonacci II/II

(I.H.) $T(n) \leq a\phi^n - b$

Substitute the I.H.

$$T(n) \leq a(\phi^{n-1} + \phi^{n-2}) - 2b + \Theta(1)$$

# Work of Parallel Fibonacci II/II

(I.H.) $\quad T(n) \leq a\phi^n - b$

Substitute the I.H.

$$
\begin{aligned}
T(n) &\leq a(\phi^{n-1} + \phi^{n-2}) - 2b + \Theta(1) \\
&= a\frac{\phi + 1}{\phi^2}\phi^n - b + (\Theta(1) - b)
\end{aligned}
$$

# Work of Parallel Fibonacci II/II

(I.H.) $T(n) \leq a\phi^n - b$

Substitute the I.H.

$$\begin{aligned} T(n) &\leq a(\phi^{n-1} + \phi^{n-2}) - 2b + \Theta(1) \\ &= a\frac{\phi + 1}{\phi^2}\phi^n - b + (\Theta(1) - b) \end{aligned}$$

for $b$ sufficiently large

$$\leq a\frac{\phi + 1}{\phi^2}\phi^n - b$$

# Work of Parallel Fibonacci II/II

(I.H.) $\quad T(n) \leq a\phi^n - b$

Substitute the I.H.

$$\begin{aligned} T(n) &\leq a(\phi^{n-1} + \phi^{n-2}) - 2b + \Theta(1) \\ &= a\frac{\phi+1}{\phi^2}\phi^n - b + (\Theta(1) - b) \end{aligned}$$

for $b$ sufficiently large

$$\leq a\frac{\phi+1}{\phi^2}\phi^n - b = a\phi^n - b$$

# Span and Parallelism of Fibonacci

$$\begin{aligned} T_\infty(n) &= \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1) \\ &= T_\infty(n-1) + \Theta(1) \end{aligned}$$

# Span and Parallelism of Fibonacci

$$
\begin{aligned}
T_\infty(n) &= \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1) \\
&= T_\infty(n-1) + \Theta(1)
\end{aligned}
$$

Transforming to sum, we get

$$T_\infty \in \Theta(n)$$

# Span and Parallelism of Fibonacci

$$\begin{aligned} T_\infty(n) &= \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1) \\ &= T_\infty(n-1) + \Theta(1) \end{aligned}$$

Transforming to sum, we get

$$T_\infty \in \Theta(n)$$

$$\text{parallelism} = \frac{T_1(n)}{T_\infty(n)} = \Theta\left(\frac{\phi^n}{n}\right)$$

# Span and Parallelism of Fibonacci

$$T_\infty(n) = \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1)$$
$$= T_\infty(n-1) + \Theta(1)$$

Transforming to sum, we get

$$T_\infty \in \Theta(n)$$

$$\text{parallelism} = \frac{T_1(n)}{T_\infty(n)} = \Theta\left(\frac{\phi^n}{n}\right)$$

- inefficient, but very parallel

# Parallel Loops

**parallel for** $i = 1$ to $n$ **do**
  statement...
  statement...
**end for**

- Run $n$ copies in parallel with local setting of $i$.

# Parallel Loops

**parallel for** $i = 1$ to $n$ **do**
  statement...
  statement...
**end for**

- Run $n$ copies in parallel with local setting of $i$.
- Effectively $n$-way spawn

# Parallel Loops

**parallel for** $i = 1$ to $n$ **do**
    statement...
    statement...
**end for**

- Run $n$ copies in parallel with local setting of $i$.
- Effectively $n$-way spawn
- Can be implemented with spawn and sync

# Parallel Loops

**parallel for** $i = 1$ to $n$ **do**
  statement...
  statement...
**end for**

- Run $n$ copies in parallel with local setting of $i$.
- Effectively $n$-way spawn
- Can be implemented with spawn and sync
- Span

$$T_\infty(n) = \Theta(\log n) + \max_i T_\infty(\text{iteration i})$$

# Parallel Matrix-Vector product

To compute $y = Ax$

$$y_i = \sum_{j=1}^{n} a_{ij} x_j$$

# Parallel Matrix-Vector product

To compute $y = Ax$

$$y_i = \sum_{j=1}^{n} a_{ij} x_j$$

**function** $\textsc{RowMult}$(A,x,y,i)
  $y_i = 0$
  **for** $j = 1$ to $n$ **do**
    $y_i = y_i + a_{ij} x_j$
  **end for**
**end function**

# Parallel Matrix-Vector product

To compute $y = Ax$

$$y_i = \sum_{j=1}^{n} a_{ij} x_j$$

**function** $\textsc{RowMult}$(A,x,y,i)
    $y_i = 0$
    **for** $j = 1$ to $n$ **do**
        $y_i = y_i + a_{ij} x_j$
    **end for**
**end function**

**function** $\textsc{Mat-Vec}(A, x, y)$
    Let $n =$ rows$(A)$
    **parallel for** $i = 1$ to $n$ **do**
        RowMult(A,x,y,i)
    **end for**
**end function**

# Parallel Matrix-Vector product

To compute $y = Ax$

$$y_i = \sum_{j=1}^{n} a_{ij} x_j$$

**function** $\textsc{RowMult}$(A,x,y,i)
  $y_i = 0$
  **for** $j = 1$ to $n$ **do**
    $y_i = y_i + a_{ij} x_j$
  **end for**
**end function**

$$T_1(n) \in \Theta(n^2)$$

$$T_\infty(n) = \underbrace{\Theta(\log(n))}_{\text{parallel for}} + \underbrace{\Theta(n)}_{\text{RowMult}}$$

# Parallel Matrix-Vector product

```
function RowMult(A,x,y,i)
    y_i = 0
    for j = 1 to n do
        y_i = y_i + a_ij x_j
    end for
end function
```

- Why is RowMult not using parallel for?

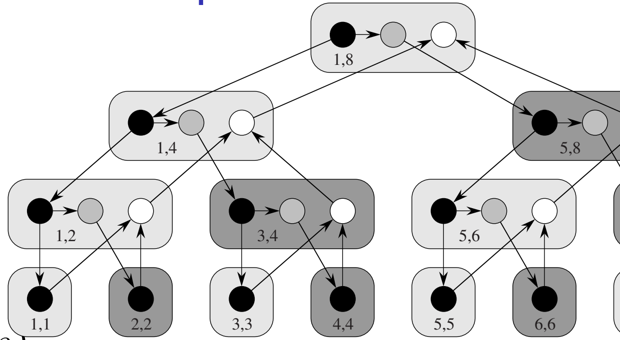
```
function Mat-Vec(A, x, y)
    Let n = rows(A)
    parallel for i = 1 to n do
        RowMult(A,x,y,i)
    end for
end function
```

# OpenMP Matrix-Vector Product

```
void MatVec(const mat &A,const vec &x,vec &y){
#pragma omp parallel for
  for(int i=0; i<A.size(); i++){
    RowMult(A, x, y, i);
  }
}
```

# Divide and Conquer Matrix-Vector product

**function** $\mathrm{MVDC}(A, x, y, f, t)$
  **if** $f == t$ **then**
    RowMult(A,x,y,f)
  **else**
    $m = \lfloor (f+t)/2 \rfloor$
    **spawn** MVDC$(A, x, y, f, m)$
    MVDC$(A, x, y, m+1, t)$
    **sync**
  **end if**
**end function**



# Divide and Conquer Matrix-Vector product

**function** $\mathrm{MVDC}(A, x, y, f, t)$
  **if** $f == t$ **then**
    RowMult(A,x,y,f)
  **else**
    $m = \lfloor (f+t)/2 \rfloor$
    **spawn** MVDC$(A, x, y, f, m)$
    MVDC$(A, x, y, m+1, t)$
    **sync**
  **end if**
**end function**

- $T_\infty(n) = \Theta(\log n) + T_\infty(\text{RowMult})$

# Divide and Conquer Matrix-Vector product

```
function MVDC(A, x, y, f, t)
    if f == t then
        RowMult(A,x,y,f)
    else
        m = ⌊(f + t)/2⌋
        spawn MVDC(A, x, y, f, m)
        MVDC(A, x, y, m + 1, t)
        sync
    end if
end function
```

- $T_\infty(n) = \Theta(\log n) + T_\infty(\text{RowMult})$
- $\Theta(n)$ leaves (one per row)

# Divide and Conquer Matrix-Vector product

```
function MVDC(A, x, y, f, t)
    if f == t then
        RowMult(A,x,y,f)
    else
        m = ⌊(f + t)/2⌋
        spawn MVDC(A, x, y, f, m)
        MVDC(A, x, y, m + 1, t)
        sync
    end if
end function
```

- $T_\infty(n) = \Theta(\log n) + T_\infty(\text{RowMult})$
- $\Theta(n)$ leaves (one per row)
- $\Theta(n)$ interior nodes (binary tree)

# Divide and Conquer Matrix-Vector product

```
function MVDC(A, x, y, f, t)
    if f == t then
        RowMult(A,x,y,f)
    else
        m = ⌊(f + t)/2⌋
        spawn MVDC(A, x, y, f, m)
        MVDC(A, x, y, m + 1, t)
        sync
    end if
end function
```

- $T_\infty(n) = \Theta(\log n) + T_\infty(\text{RowMult})$
- $\Theta(n)$ leaves (one per row)
- $\Theta(n)$ interior nodes (binary tree)
- $T_1(n) = \Theta(n^2)$

# Divide and Conquer Matrix-Vector (OpenMP)

```
void MVDC(const mat &A, const vec &x, vec &y,
          int f, int t) {
  if (f == t) {
    RowMult(A, x, y, f);
  } else {
    int m = (f+t)/2;
#pragma omp task
    MVDC(A,x,y,f,m);
    MVDC(A,x,y,m+1,t);
#pragma omp taskwait
  }
}
```