

---

# ***High Performance Computing***

## ***Lecture 20 - FFT wrap-up***

Ole Nielsen

# *Today's Outline*

---

- Parallel Performance of the FFT
- Amdahls Law
- Steps towards a 2D transform

## *FFT Timings: $N = 2^{20}$*

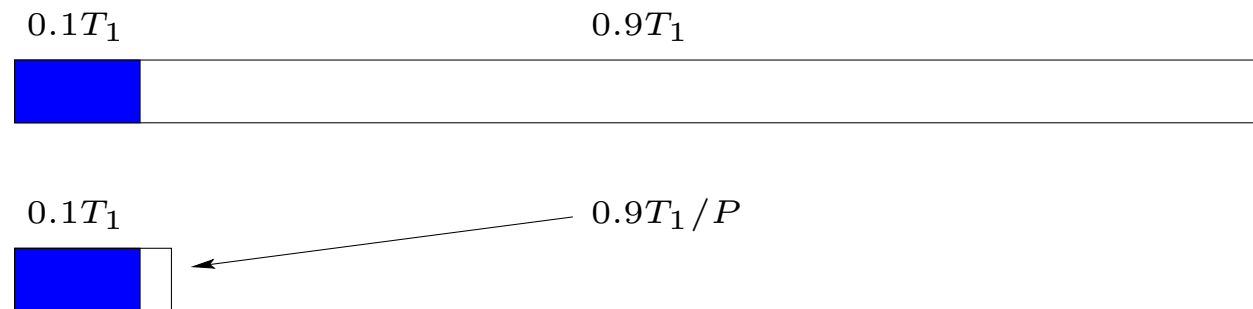
<b>Algorithm</b>	<b>Time</b>	<b>Speedup</b>
DFT	$\approx 910$ h	
FFT (loop)	27s	
FFT (vectors)	12s	
Parallel FFT (1 proc)	12s	
<b>Parallel FFT (8 active procs)</b>		
Wall clock	3.95	3
Without final gather	2.47	4.9
Without gather and twiddles	2.23	5.5

# Amdahl's Law

Computing the twiddle factors is an inherently sequential process

## Example:

Assume that sequential part takes up 10% of total (sequential) computation time.



Even if the remaining 90% is reduced maximally, total speedup can never exceed 10!

# Amdahl in general

Let  $r$  be the fraction which can be parallelised

$$T_P = (1 - r)T_1 + rT_1/P$$

$$S_P = \frac{T_1}{T_P} = \frac{T_1}{(1 - r)T_1 + rT_1/P} = \frac{1}{1 - r + r/P}$$

$$S_P \leq \lim_{P \rightarrow \infty} S_P = \frac{1}{1 - r}$$

## Examples:

- $r = 1 : S_P = P$
- $r = 0 : S_P = 1$
- $r = 0.5 : S_P \leq 2$
- $r = 0.9 : S_P \leq 10$

# ***FFT Performance model - computations***

---

Omit twiddlefactor computation and final gather

- $t_o$ : Time for one complex operation (+, -, \*)
- Computation time for parallel FFT is

$$3N \log_2 N t_o / P$$

# ***FFT Performance model - communication***

---

- $t_l$ : Latency of communications
- $t_c$ : the time it takes to communicate one complex number (16 bytes)

## **Communication model for $n$ complex numbers**

$$T = t_l + nt_b$$

There are  $\log_2 P$  levels of communication and each processor must send and receive  $N/P$  complex numbers at each level

# FFT Performance Model

---

**Total predicted execution time:**

$$T_P(N) = \underbrace{\frac{3N \log_2 N}{P} t_o}_{\text{Computation time}} + \underbrace{2 \log_2 P \left( t_l + \frac{N}{P} t_b \right)}_{\text{Communication time}}$$

This model can be verified empirically through timings of the FFT program using measured values for  $t_l$ ,  $t_b$ , and  $t_o$ .

# Steps towards a 2 Dimensional FFT

---

## 2D (tensor) $M \times N$ DFT

$$g_{m,n} = \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} f_{k,l} w_M^{mk} w_N^{nl}, \quad m = 0, \dots, M-1, n = 0, \dots, N-1$$

## Separation

$$g_{m,n} = \sum_{k=0}^{M-1} \left( \sum_{l=0}^{N-1} f_{k,l} w_N^{nl} \right) w_M^{mk}$$

# Matrix formulation

---

## 1D DFT/FFT

$$g = Ff, \quad [F]_{kl} = w^{kl}$$

Transform all columns of  $M \times N$  matrix  $X$

$$FX$$

Transform all rows of  $M \times N$  matrix  $X$

$$(FX^T)^T = XF^T$$

## 2D FFT

$$FXF^T$$

# *Sequential 2D implementation*

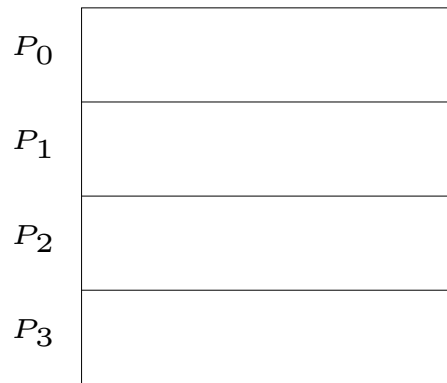
---

```
def fft2(X):  
  
    M, N = X.shape  
  
    # For each row  
    for i in range(M):  
        X[i,:] = fft(X[i,:])  
  
    # For each column  
    for j in range(N):  
        X[:,j] = fft(X[:,j])  
  
    return X
```

# Parallel 2D implementation

---

## Data distribution



## Row transforms in parallel:

```
# For each row
for i in range(i0,i1):
    X[i,:] = fft(X[i,:])
```

## Column transforms in parallel:

Reuse code from 1D parallel FFT and replicate across all columns using vector operations.

# Improvements

---

- Reuse twiddle factors
- Use vector operations for the multiple (both row and column) transforms instead of calling the 1D fft many times.

## Example: Replicate a calculation across columns

- **1D transform:**  $XR = X[r0:r1]*w$
- **All columns:**  $XR = X[r0:r1, :]*w$
- **Some columns:**  $XR = X[r0:r1, c0:c1]*w$

# Conclusion

---

- Good speedup achievable for parallel FFT
- Excellent speedup if twiddle factors and final gather omitted
- Multiple FFT's would be very feasible (twiddle factors reused reducing effect of Amdahl)
- Final gather unnecessary if FFT is part of a bigger application, e.g. PDE solver
- 2D FFT's will be very feasible due to higher ratio between computations and communications

# *Today's exercise*

---

- Finish the parallel FFT
- Assess the speedup
- Optional: Verify performance model