Parallel Fourier Transforms

Iain Bethune, Gavin Pringle, Joachim Hein
EPCC
The University of Edinburgh

Overview

- Fourier Transforms and Fast Fourier Transforms
- Parallel FFT in 1 Dimension – shared memory
- Parallel Fourier Transformations of 2D arrays
- Intro to FFTs of 3D arrays
- Example application: large integer multiplication

Fourier Transform

- Fourier Transforms are important and useful mathematical operation

$$\tilde{f}(k) = \sum_{x=1}^{N} f(x) e^{\frac{-ik.x}{2\pi}} \qquad \forall\, k \in \{1 \cdots N\}$$

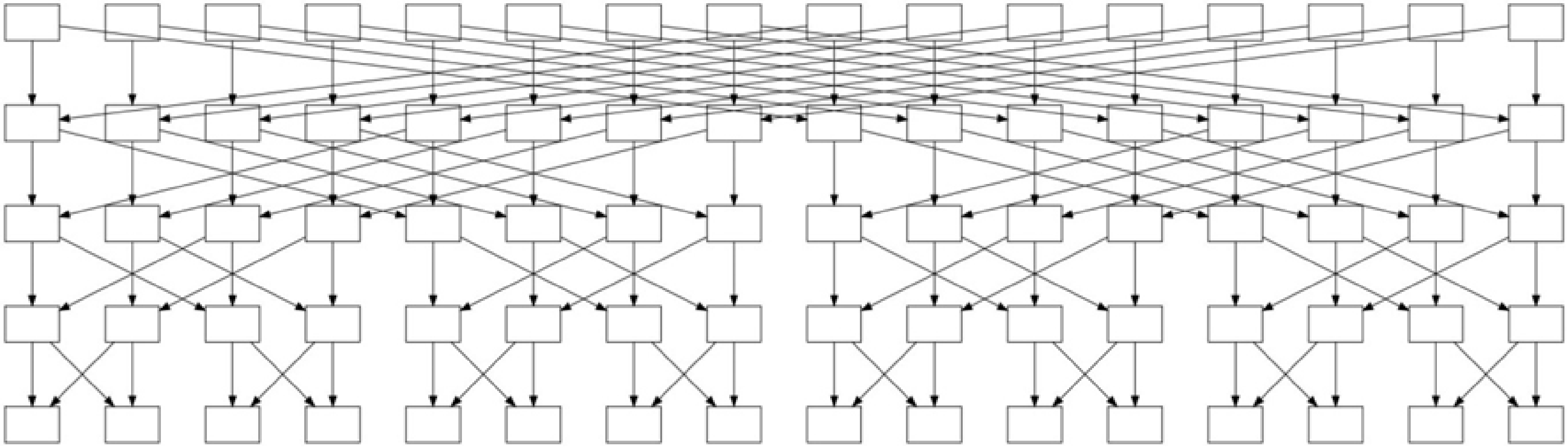
  - Every word of output depends on every word of input
  - Implemented simply has $O(N^2)$ complexity
- Fast Fourier Transform (FFT)
  - Very clever class of implementation algorithms $O\big(N.\,log(N)\big)$ operations
  - Cooley Tukey 1965 though basic approach goes back to Gauss 1805

Fast Fourier Transformation

- FFTs are often “the” critical bottleneck
  - Still a lot of data movement in FFT. This is often the limiting factor not the number of floating point operations.
  - preventing parallel application from scaling to larger numbers of processors due to communications

- This lecture discusses reasons and how we might parallelise an FFT to overcome this

FFT algorithm



- Parallel nature best understood in graphical form.
  - O(N log(N)) floating point complexity.
  - O(N) potential parallelism with good load balance.
  - Non-local operations imply significant communication. Performance of parallel implementations tends to be limited by communication/memory-traffic.
  - Longer range interaction have to be performed first.

Parallel 1D FFT

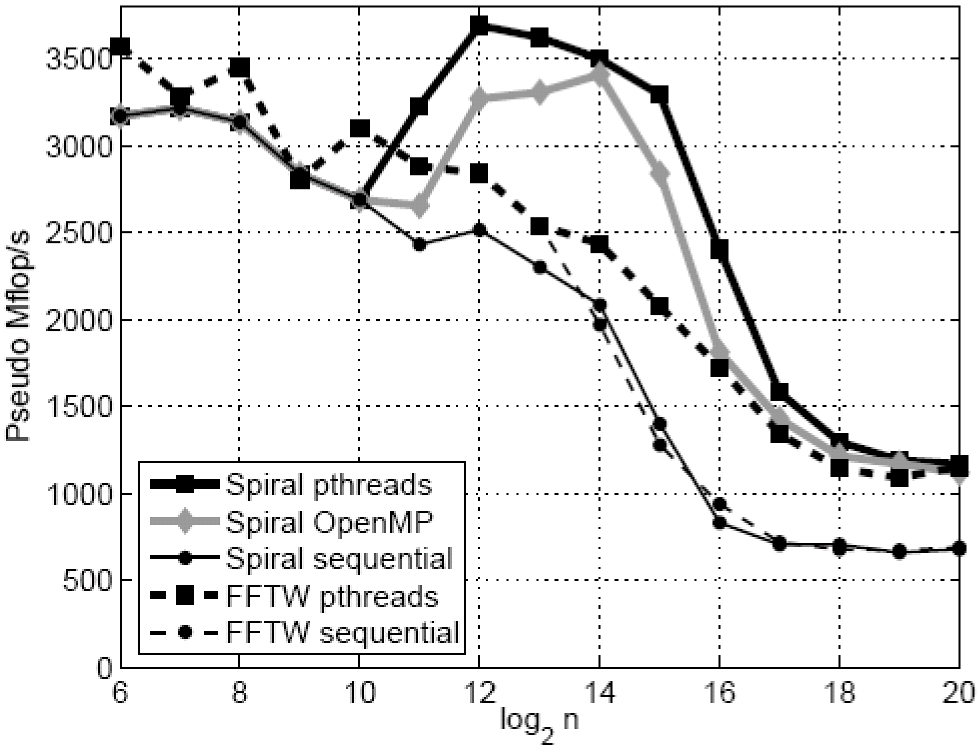
- Parallelisation of a 1D FFT is hard
  - Somewhat easier in shared memory but still communication (memory bandwidth) limited
  - In practice use one of the many high quality numerical library implementations that support thread parallelism.
- Algorithm is hard to decompose without shared memory
  - Distributed approaches usually rely on converting 1D FFT into a modified 2D FFT and decomposing that (see later).
  - Only worthwhile if N very large.
- Typically **`N≈100–200`** in many scientific codes e.g. materials chemistry – small amount of data
- Literature examples:

  Franchetti, Voronenko, Püschel, "FFT Program Generation for Shared Memory: SMP and Multicore", Paper presented at SC06, Tampa, FL http://sc06.supercomputing.org/schedule/pdf/pap169.pdf

  Tang et al, "A Framework for Low-Communication 1-D FFT", SC12, http://blogs.intel.com/intellabs/files/2012/11/SC12_FFT-pap147s4-final.pdf

“Traditional” SMP

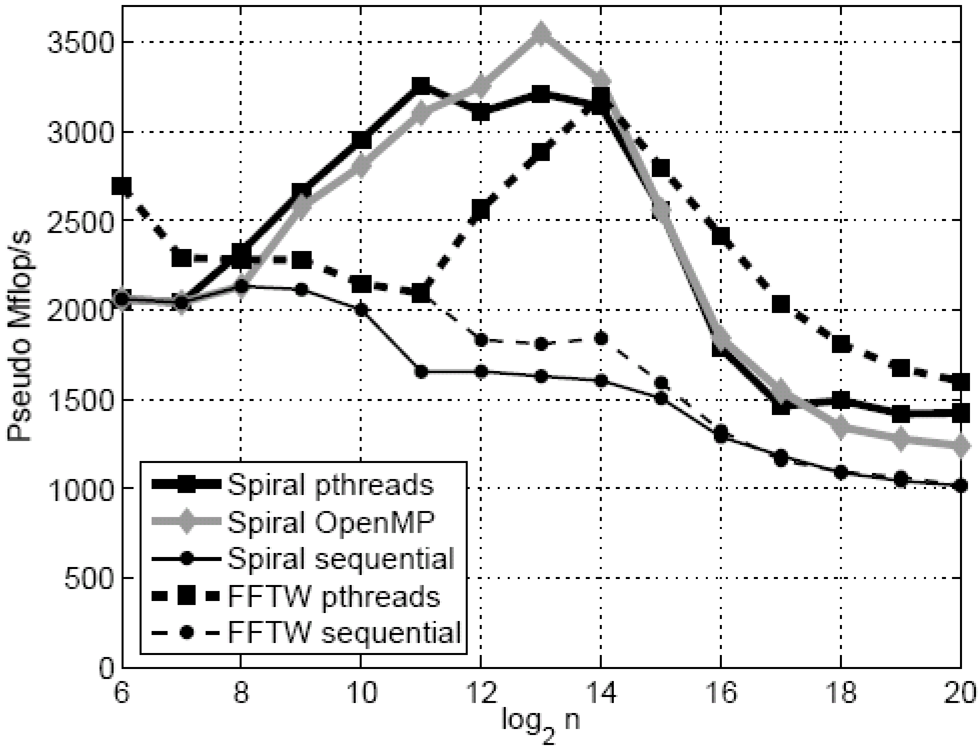
- 4 processor Intel Xeon
  - Communication via shared Memory (Bus)
- Benefits from:
  - N=2048 (Spiral)
  - N=16384 (FFTW)
- Improvement for large problems (Factor about 2 for 4 CPU)



(d) 2.8 GHz Xeon MP (4 processors)

Multicore Processor

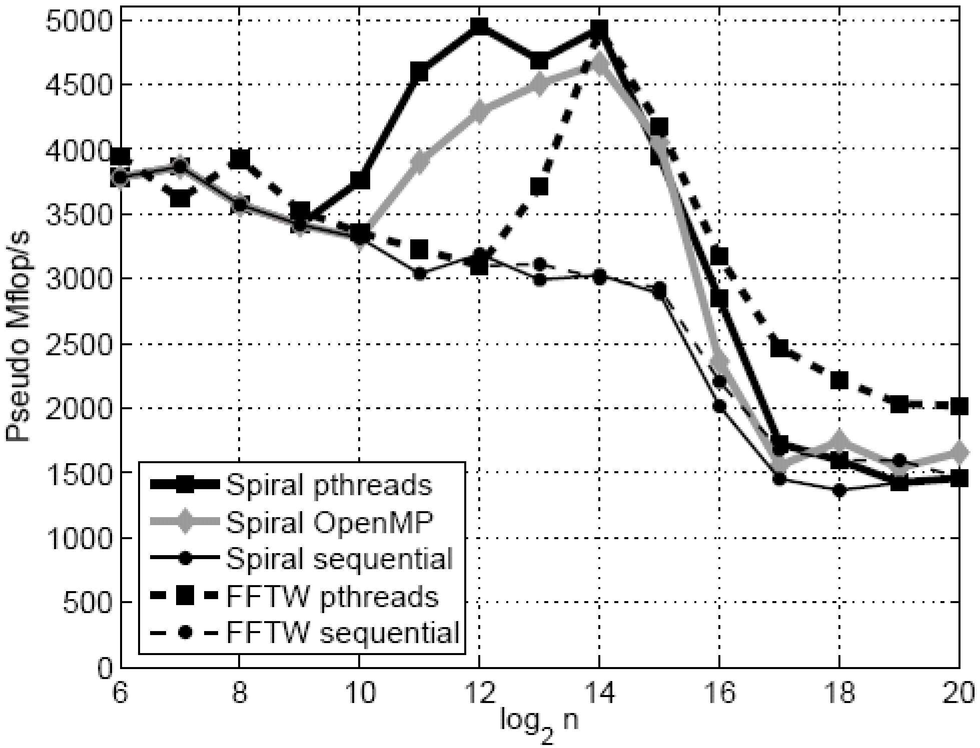
- Intel Core Duo (laptop)
  - Shared L2 used for communication
- Benefits from
  - N=512 (Spiral)
  - N=4096 (FFTW)
- Not as efficient for huge problems



(a) 2.0 GHz Core Duo (2 processors)

Multicore processor with shared bus

- Intel Pentium D
  - Multicore chip
  - Communication via Bus
- Benefits from:
  - N=2048 (Spiral)
  - N=8192 (FFTW)
- Little benefit for huge problems (shared bus)



(c) 3.6 GHz Pentium D (2 processors)

Summary: 1D parallel FFT

- Parallelisation works for large problems only ☹
- Sensitive to contention (shared buses) ☹
- Multicore chips with communications at cache level appear beneficial – might "be there" in a few years time
- Shows speedup, but not always "perfect" 😐

FFTs in two dimensions

- What needs calculating for a 2D FFT:

$$\tilde{f}(k,l) = \sum_{y=1}^{M} \left\{ \sum_{x=1}^{N} \left[f(x,y) \exp\left(-2\pi \mathrm{i}\frac{kx}{N}\right)\right] \exp\left(-2\pi \mathrm{i}\frac{ly}{M}\right)\right\}$$

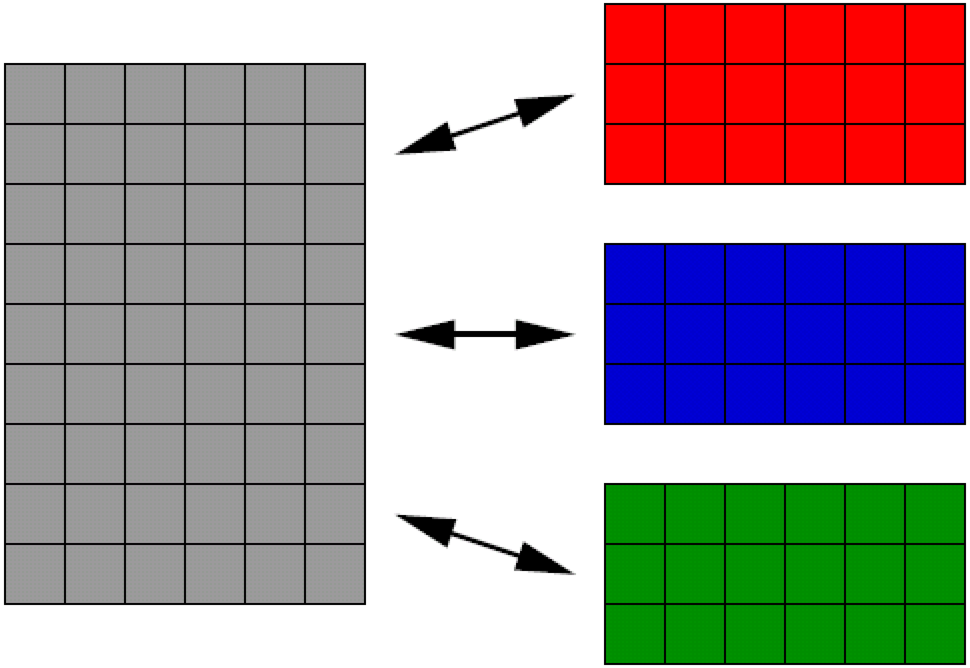
- Do it in a 2 step approach:

$$\hat{f}(k,y) \equiv \sum_{x=1}^{N} \left[f(x,y) \exp\left(-2\pi \mathrm{i}\frac{kx}{N}\right)\right]$$

$$\tilde{f}(k,l) = \sum_{y=1}^{M} \left\{ \hat{f}(k,y) \exp\left(-2\pi \mathrm{i}\frac{ly}{M}\right)\right\}$$

Distribute array onto 1D processor grid

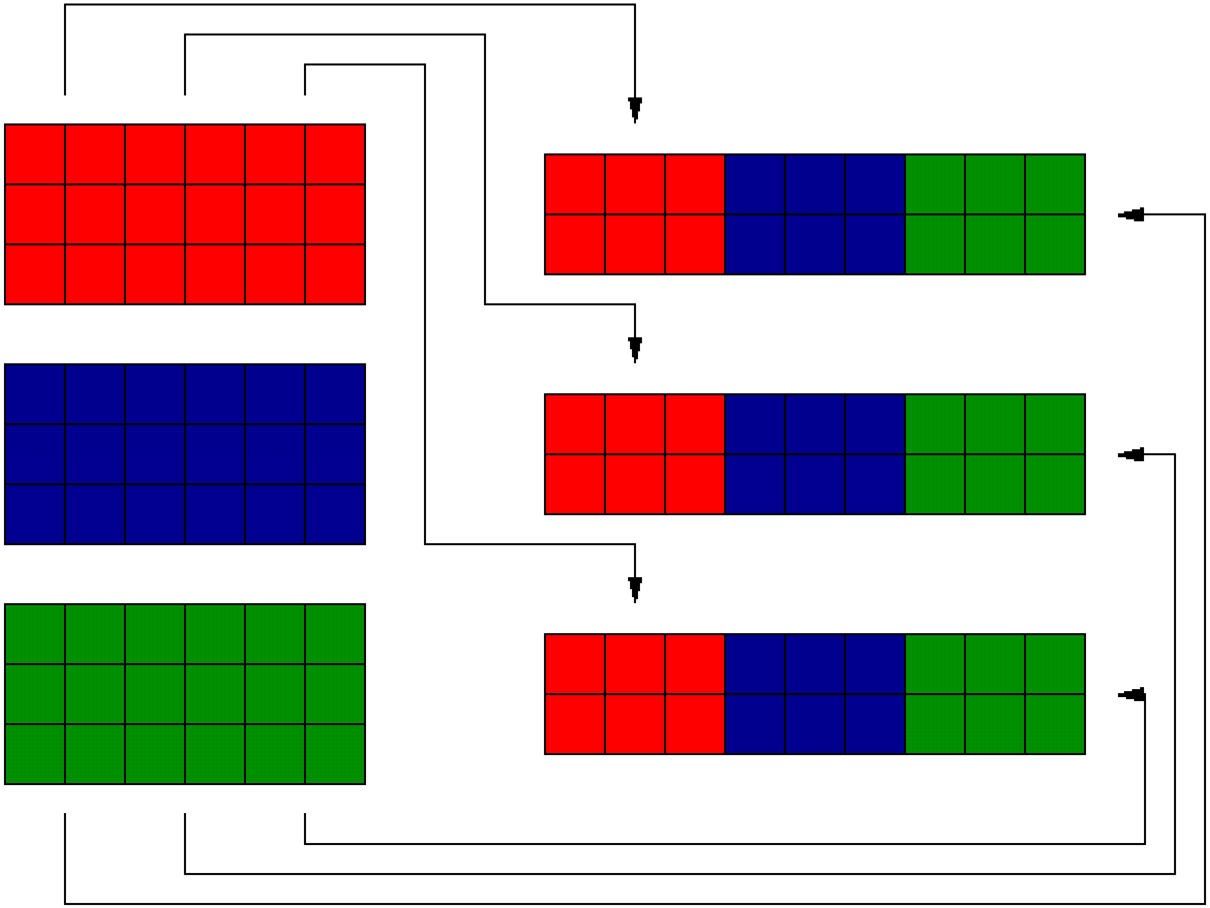
- Example:
  - 6×9 array
  - 3 processors
  - Assuming row major order (C convention)



- Perform 1st FFT:
  - Each processor transforms 3 arrays of 6 elements
- What next?

Data transposition

- Divide up the array by columns for 2nd FFT
- Depending on FFT library simultaneous transpose can be advantageous (shown on figure)



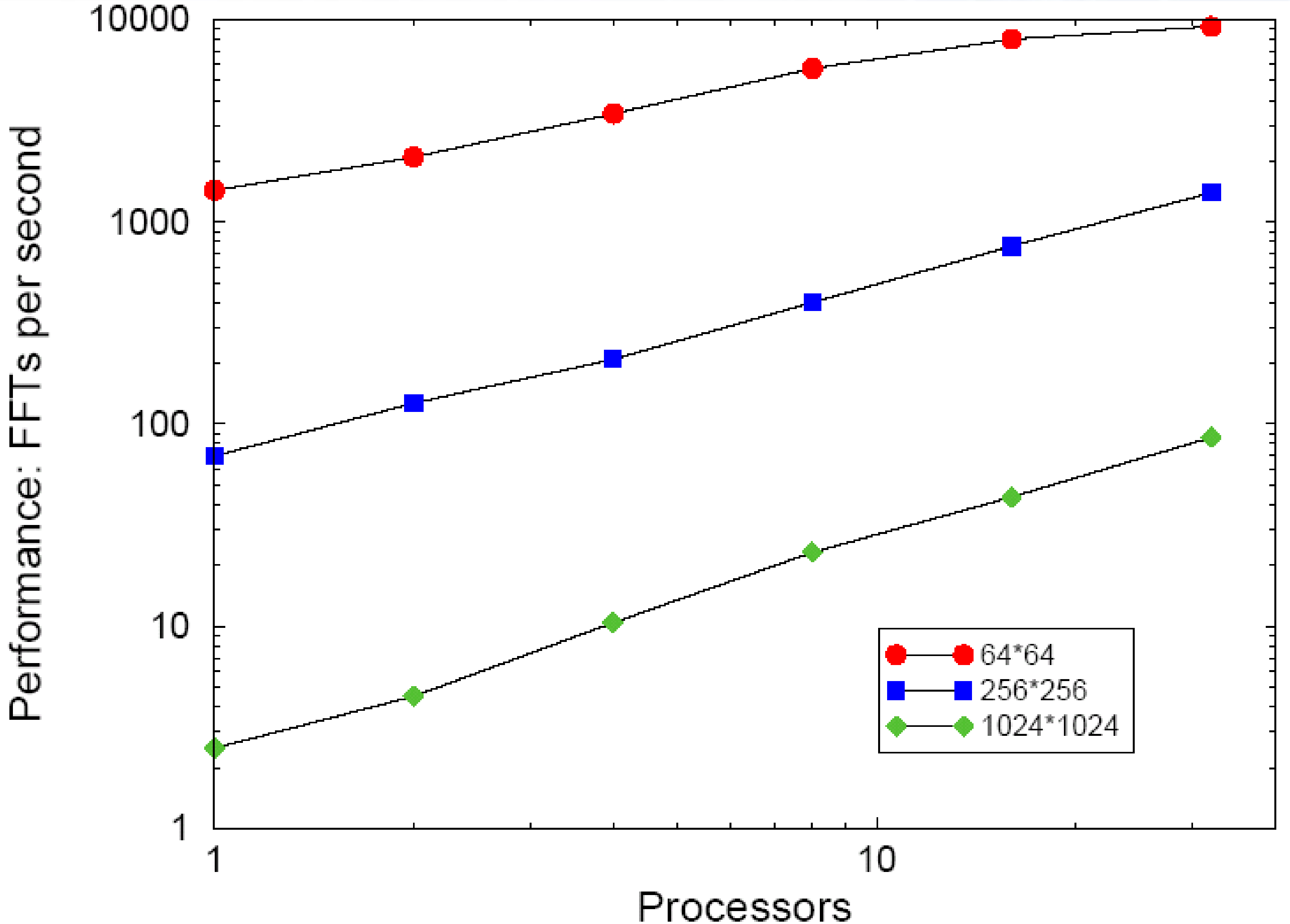
Perform 2nd FFT

- What used to be the columns of the original array is now in row-major order ☺

- Do the 2nd FFT
  - In the example:
    Each processor performs 2 FFTs of an array of length 9

- Rearrange data as required by following code
  - Examples:
    - Undoing the transpose
    - Redistributing data onto 2D grid
    - Sometimes: nothing needs done ☺

Example 2D-FFT on 32 BlueGene/L CPUs



Fourier Transformation of a 3D array

- Definition of the Fourier Transformation of a three dimensional array $\boldsymbol{A_{x,y,z}}$

$$\tilde{A}_{u,v,w} :=$$

$$\underbrace{\sum_{x=0}^{L-1} \underbrace{\sum_{y=0}^{M-1} \underbrace{\sum_{z=0}^{N-1} A_{x,y,z} \exp(-2\pi \mathrm{i} \frac{wz}{N})}_{\text{1st 1D FT along } z} \exp(-2\pi \mathrm{i} \frac{vy}{M})}_{\text{2nd 1D FT along } y} \exp(-2\pi \mathrm{i} \frac{ux}{L})}_{\text{3rd 1D FT along } x}$$

- Can be performed as three subsequent 1 dimensional Fourier Transformations

Parallel 3D FFT

- Below N processors only need to decompose in 1D and use 1 transpose
  - One dimension always local
  - “slab” decomposition
- Above N processors Can decompose in 2 D and use 2 transposes.
  - “pencil” decomposition
  - Two transposes double the communication cost
  - Communication is frequently most of the cost anyway.
  - FFT width can impose a scaling limit on many codes.

Large Integer Multiplication

- Some application areas e.g. computational number theory need to store and operate on very large integers
  - e.g. largest known prime number has 17.4 million decimal digits

- What representation should we use?

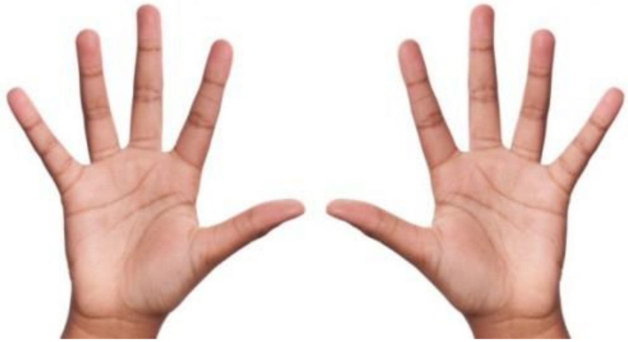
- Integers
  - Precisely represent whole numbers
  - Range limitation: 32 bits = ~4 billion, 64 bits = 1.8×10^{19}

- Floating point
  - Range still not enough (s.p. = $\sim 10^{38}$, d.p. = $\sim 10^{308}$)
  - Only finite precision (8 or 16 sig. fig.)

Large Integer Multiplication

- Need to define an arbitrary-sized large integer format

- We work in base 10



- Numbers larger than 10 we expand as a series in powers of the base e.g.
  - $123456 = 1 \times 10^5 + 2 \times 10^4 + 3 \times 10^3 + 4 \times 10^2 + 5 \times 10^1 + 6 \times 10^0$

- In general we can write any integer `a` in some base `b` as:
  - $a = a_0 * b^0 + a_1 \times b^1 + a_2 \times b^2 \ldots a_{n-1} \times b^{n-1}$

- So we can just store the terms $a_0 \ldots a_{n-1}$ in an array

Large Integer Multiplication

- Free to choose an appropriate storage format and base:
  - Want a base so that no arithmetic operations cause overflow (or loss of precision)
  - Using 64-bit integers, `b` could be 2^{32} = ~ 4 billion
  - Using double-precision IEEE floats, `b` could be 2^{26} = ~ 67 million

- Example operations:
  - Addition
  - Multiplication (or Squaring)

Large Integer Multiplication

- Addition of two large integers is ‘easy’ and fast (O(`n`))

```
c = a + b:

    carry = 0
    for i = 0 .. n-1
     ci = ai + bi
     ci = ci + carry
     carry = ci / base
     ci = ci - base * carry
```

- Check it for yourself

Large Integer Multiplication

```
        1 2
x       3 4
-----------
      2 x 4
    1 0 x 4
    2 x 3 0
  1 0 x 3 0
-----------
          8
        4 0
        6 0
      1 0 0
-----------
```

- We compute the multiple of each digit with every other digit – O(n^2)
- Sum per-digit, then perform the carries
- In 1971, Schönhage and Strassen observed the analogy with the convolution operation (e.g. in signal processing)
- Convolution can be calculated in O(p log(p)) using Fast Fourier Transform

Large Integer Multiplication

- So multiplication of two `n`-digit numbers `X` and `Y` becomes:

```
Pad X and Y to 2n digits
X' = FFT(X)
Y' = FFT(Y)
XY' = X' * Y'
```
(multiply each term)

$XY = FFT^{-1}(XY')$

- Still need to propagate carries O(n)
- Instead of doing an integer DFT, we do a floating-point FFT, and round back to integer (subject to errors)

Summary

- Parallelisation of an individual 1D FFT is hard
  - Presently works best for large problems
  - Recent advances in algorithms & hardware encouraging

- Multidimensional problems need to calculate many 1D FFTs
  - Parallelisable by distributing entire FFTs onto the processors and using a standard serial 1D FFT library
  - Requires redistributing the data between FFT dimensions

- FFT can be used to reduce computational complexity of convolution from $O(n^2)$ to $O(n \log(n))$
  - Applications in signal processing and large integer arithmetic