

---

---

**Parallel Spectral Methods:  
Fast Fourier Transform (FFT)  
with Applications**

**James Demmel**

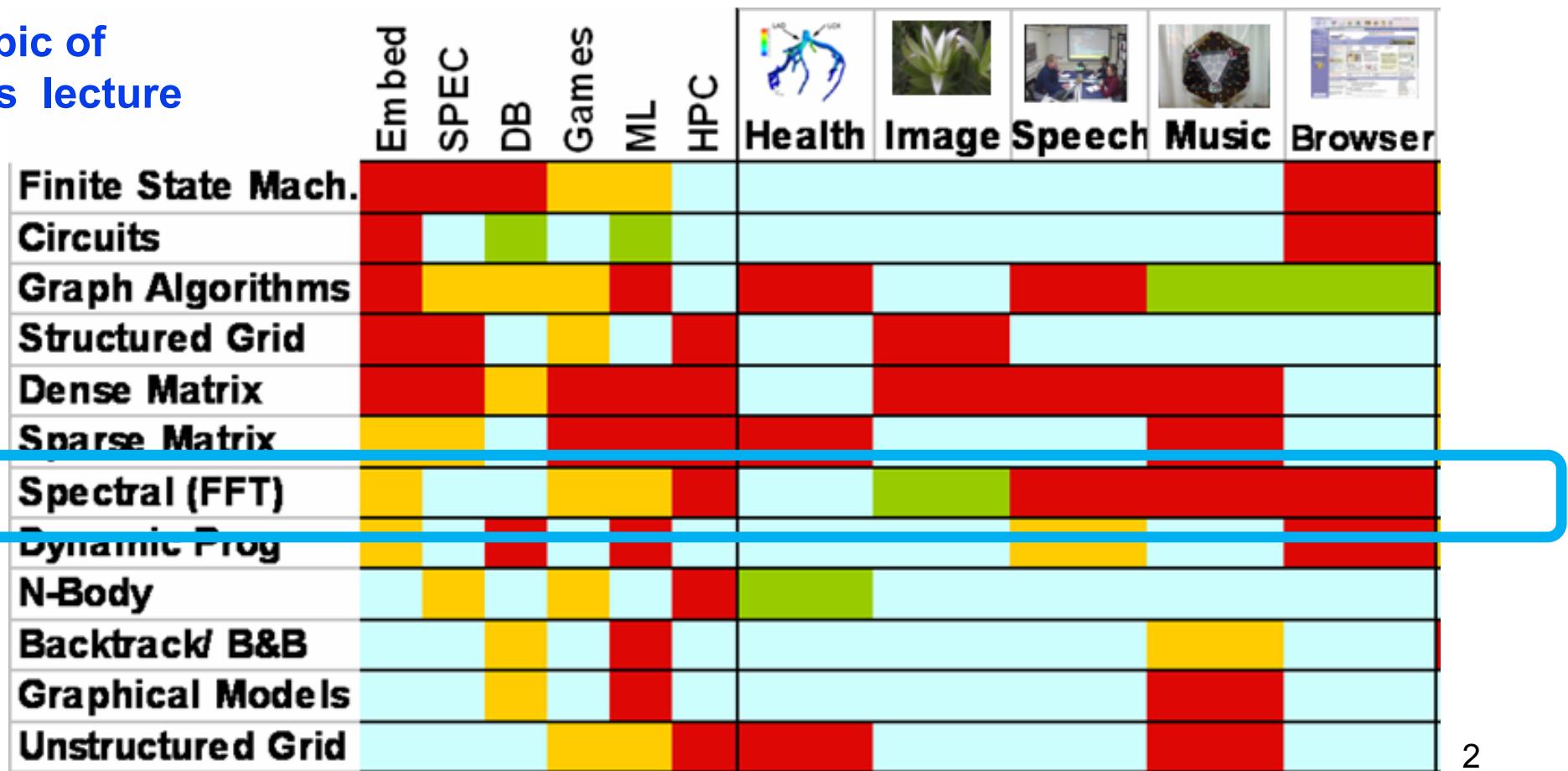
**[www.cs.berkeley.edu/~demmel](http://www.cs.berkeley.edu/~demmel)**

# Motifs

---

The Motifs (formerly “Dwarfs”) from  
“The Berkeley View” (Asanovic et al.)  
**Motifs** form key computational patterns

Topic of  
this lecture



# Ouline and References

---

- **Outline**
  - Definitions
  - A few applications of FFTs
  - Sequential algorithm
  - Parallel 1D FFT
  - Parallel 3D FFT
  - Autotuning FFTs: FFTW and Spiral projects
- **References**
  - Previous CS267 lectures
  - FFTW project: <http://www.fftw.org>
  - Spiral project: <http://www.spiral.net>
  - LogP: UCB EECS Tech Report UCB/CSD-92-713
  - Lecture by Geoffrey Fox:  
<http://grids.ucs.indiana.edu/ptliupages/presentations/PC2007/cps615fft00.ppt>

## Definition of Discrete Fourier Transform (DFT)

---

- ° Let  $i = \sqrt{-1}$  and index matrices and vectors from 0.
- ° The (1D) DFT of an m-element vector  $v$  is:

$$F^*v$$

where  $F$  is an m-by-m matrix defined as:

$$F[j,k] = \omega^{(j*k)}, \quad 0 \leq j, k \leq m-1$$

and where  $\omega$  is:

$$\omega = e^{(2\pi i/m)} = \cos(2\pi/m) + i * \sin(2\pi/m)$$

$\omega$  is a complex number with whose  $m^{\text{th}}$  power  $\omega^m = 1$  and is therefore called an  $m^{\text{th}}$  root of unity

- ° E.g., for  $m = 4$ :  $\omega = i$ ,  $\omega^2 = -1$ ,  $\omega^3 = -i$ ,  $\omega^4 = 1$
- ° The 2D DFT of an m-by-m matrix  $V$  is  $F^*V^*F$ 
  - Do 1D DFT on all the columns independently, then all the rows
- ° Higher dimensional DFTs are analogous

# Motivation for *Fast Fourier Transform (FFT)*

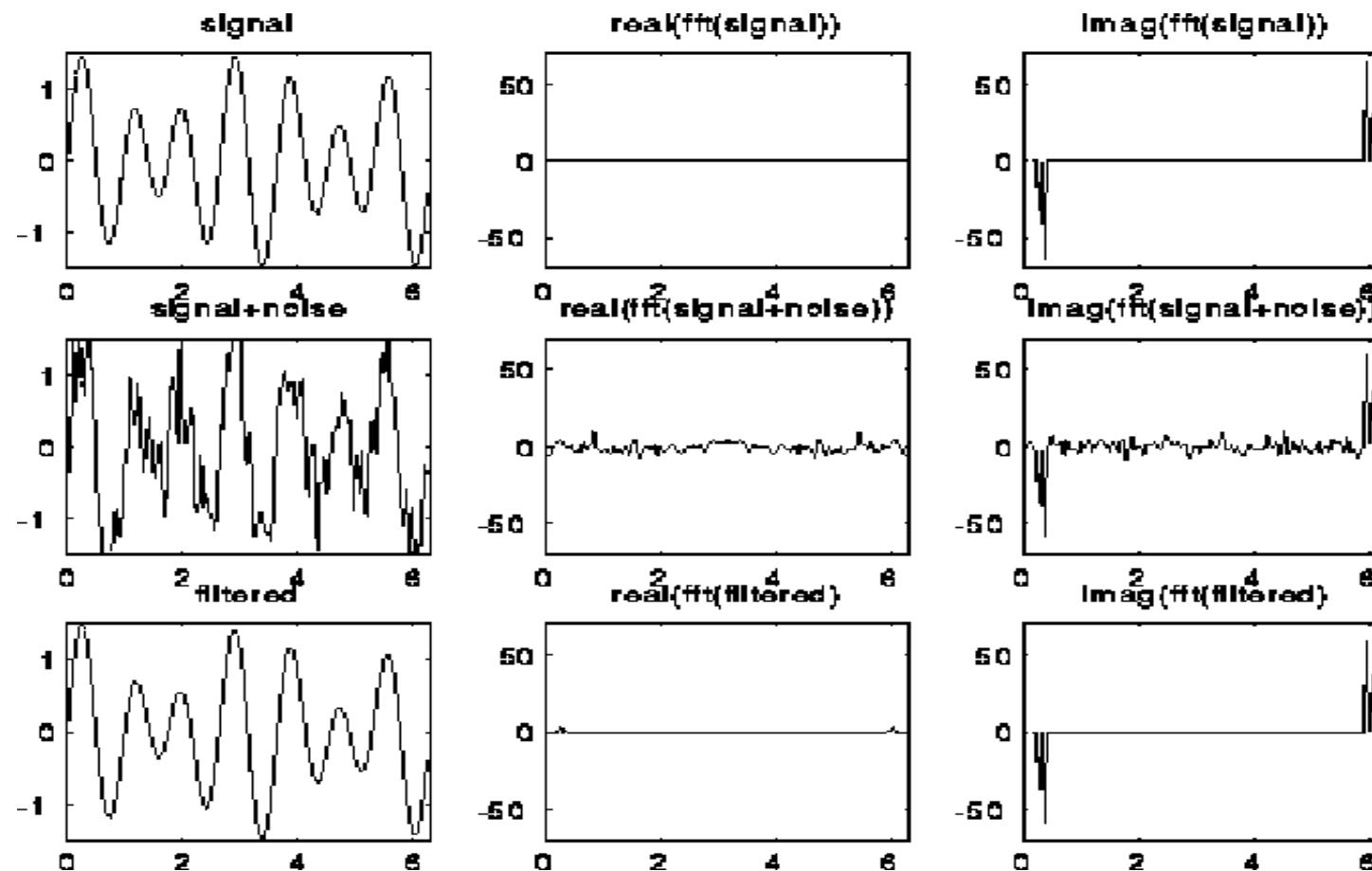
---

- **Signal processing**
- **Image processing**
- **Solving Poisson's Equation nearly optimally**
  - $O(N \log N)$  arithmetic operations,  $N = \# \text{unknowns}$
  - Competitive with multigrid
- **Fast multiplication of large integers**
  - Schonhage-Strassen:  $O(b \cdot \log b \cdot \log \log b)$  where  $b = \# \text{bits}$
- ...

## Using the 1D FFT for filtering

---

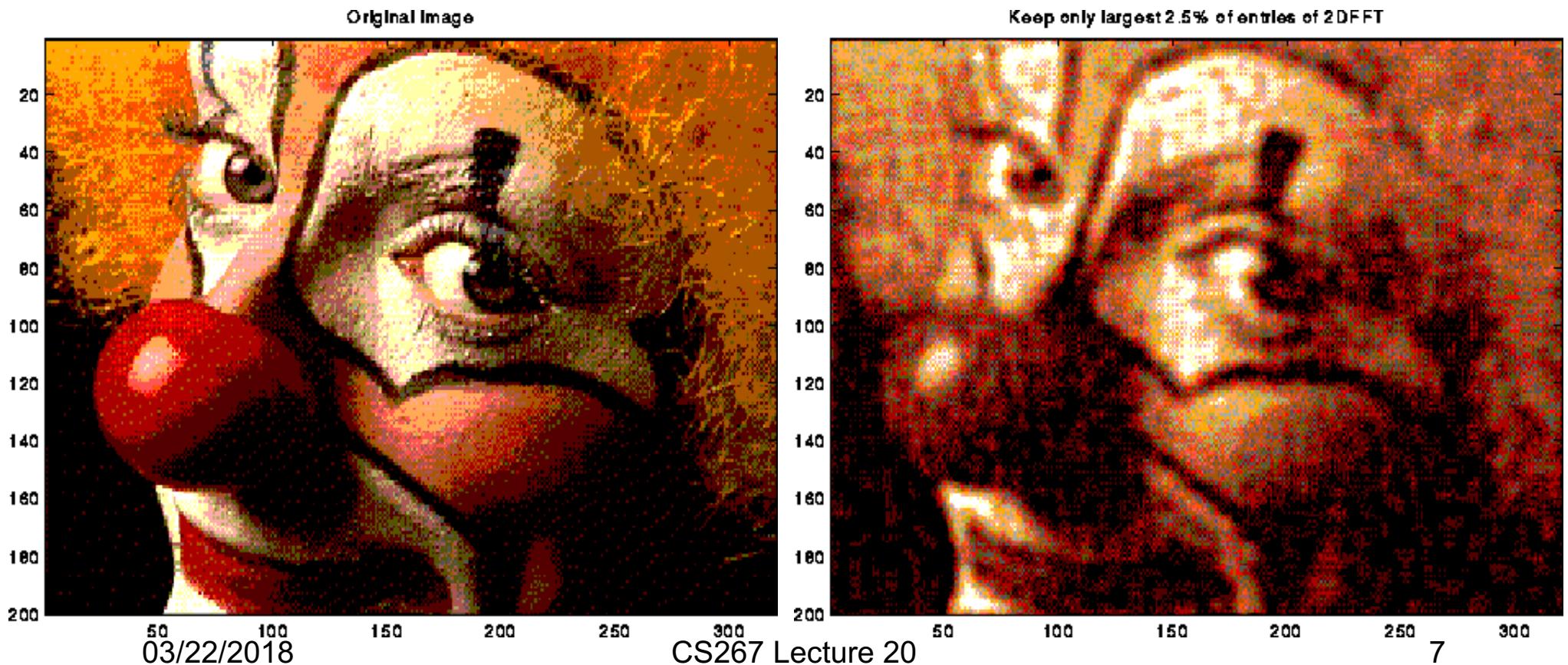
- ° Signal =  $\sin(7t) + .5 \sin(5t)$  at 128 points
- ° Noise = random number bounded by .75
- ° Filter by zeroing out FFT components < .25



# Using the 2D FFT for image compression

---

- ° **Image = 200x320 matrix of values**
- ° **Compress by keeping largest 2.5% of FFT components**
- ° **Similar idea used by jpeg**



## Recall: Poisson's equation arises in many models

---

$$3D: \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = f(x,y,z)$$

$$2D: \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y)$$

$$1D: \frac{d^2 u}{dx^2} = f(x)$$

$f$  represents the sources; also need boundary conditions

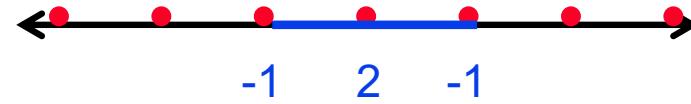
- Electrostatic or Gravitational Potential: **Potential(position)**
- Heat flow: **Temperature(position, time)**
- Diffusion: **Concentration(position, time)**
- Fluid flow: **Velocity, Pressure, Density(position, time)**
- Elasticity: **Stress, Strain(position, time)**
- Variations of Poisson have variable coefficients

# Solving Poisson Equation with FFT (1/2)

- 1D Poisson equation: solve  $L_1 x = b$  where

$$L_1 = \begin{pmatrix} 2 & -1 & & \\ -1 & 2 & -1 & \\ & -1 & 2 & -1 \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}$$

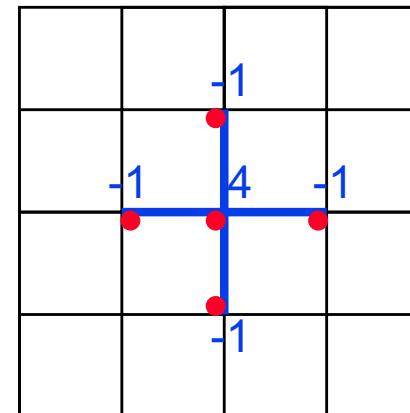
## Graph and “stencil”



- 2D Poisson equation: solve  $L_2 x = b$  where

$$L_2 = \left( \begin{array}{ccc|c|c|c} 4 & -1 & & -1 & & \\ -1 & 4 & -1 & & -1 & \\ -1 & & 4 & & & -1 \\ \hline -1 & & & 4 & -1 & -1 \\ & -1 & & -1 & 4 & -1 & -1 \\ & & -1 & & -1 & 4 & -1 \\ \hline & & & -1 & & 4 & -1 \\ & & & & -1 & 4 & -1 \\ & & & & & -1 & 4 \end{array} \right)$$

## Graph and “5 point stencil”



3D case is analogous  
(7 point stencil)

## Solving 2D Poisson Equation with FFT (2/2)

---

- **Use facts that**
  - $L_1 = F \cdot D \cdot F^T$  is eigenvalue/eigenvector decomposition, where
    - **F is very similar to FFT (imaginary part)**
      - $F(j,k) = (2/(n+1))^{1/2} \cdot \sin(j k \pi / (n+1))$
    - **D = diagonal matrix of eigenvalues**
      - $D(j,j) = 2(1 - \cos(j \pi / (n+1)))$
  - **2D Poisson same as solving  $L_1 \cdot X + X \cdot L_1 = B$  where**
    - **X square matrix of unknowns at each grid point, B square too**
- **Substitute  $L_1 = F \cdot D \cdot F^T$  into 2D Poisson to get algorithm**
  1. Perform 2D “FFT” on B to get  $B' = F^T \cdot B \cdot F$ , or  $B = F \cdot B' \cdot F^T$   
Get  $FDF^T X + XFDF^T = FB'F^T$  or  $F[D(F^T X F) + (F^T X F)D]F^T = F[B']F^T$  or  $DX' + X'D = B'$
  2. Solve  $D X' + X' D = B'$  for  $X'$ :  $X'(j,k) = B'(j,k) / (D(j,j) + D(k,k))$
  3. Perform inverse 2D “FFT” on  $X' = F^T \cdot X \cdot F$  to get  $X = F \cdot X' \cdot F^T$
- **Cost = 2 2D-FFTs plus  $n^2$  adds, divisions =  $O(n^2 \log n)$**
- **3D Poisson analogous**

# Algorithms for 2D (3D) Poisson Equation ( $N = n^2$ ( $n^3$ ) vars)

Algorithm	Serial	PRAM	Memory	#Procs
◦ Dense LU	$N^3$	$N$	$N^2$	$N^2$
◦ Band LU	$N^2$ ( $N^{7/3}$ )	$N$	$N^{3/2}$ ( $N^{5/3}$ )	$N$ ( $N^{4/3}$ )
◦ Jacobi	$N^2$ ( $N^{5/3}$ )	$N$ ( $N^{2/3}$ )	$N$	$N$
◦ Explicit Inv.	$N^2$	$\log N$	$N^2$	$N^2$
◦ Conj.Gradients	$N^{3/2}$ ( $N^{4/3}$ )	$N^{1/2(1/3)} * \log N$	$N$	$N$
◦ Red/Black SOR	$N^{3/2}$ ( $N^{4/3}$ )	$N^{1/2}$ ( $N^{1/3}$ )	$N$	$N$
◦ Sparse LU	$N^{3/2}$ ( $N^2$ )	$N^{1/2}$	$N * \log N$ ( $N^{4/3}$ )	$N$
→ ◦ FFT	$N * \log N$	$\log N$	$N$	$N$
◦ Multigrid	$N$	$\log^2 N$	$N$	$N$
◦ Lower bound	$N$	$\log N$	$N$	

PRAM is an idealized parallel model with zero cost communication

Reference: James Demmel, Applied Numerical Linear Algebra, SIAM, 1997.

## Related Transforms

---

- **Most applications require multiplication by both F and F<sup>-1</sup>**
  - $F(j,k) = \exp(2\pi i j k / m)$
- **Multiplying by F and F<sup>-1</sup> are essentially the same.**
  - $F^{-1} = \text{complex\_conjugate}(F) / m$
- **For solving the Poisson equation and various other applications, we use variations on the FFT**
  - The sin transform -- imaginary part of F
  - The cos transform -- real part of F
- **Algorithms are similar, so we will focus on F**

## Serial Algorithm for the FFT

---

- ° Compute the FFT ( $F^*v$ ) of an  $m$ -element vector  $v$

$$\begin{aligned}(F^*v)[j] &= \sum_{k=0}^{m-1} F(j,k) * v(k) \\&= \sum_{k=0}^{m-1} \varpi^{(j*k)} * v(k) \\&= \sum_{k=0}^{m-1} (\varpi^j)^k * v(k) \\&= V(\varpi^j)\end{aligned}$$

where  $V$  is defined as the polynomial

$$V(x) = \sum_{k=0}^{m-1} x^k * v(k)$$

- ° FFT same as evaluating a polynomial  $V(x)$  with degree  $m-1$  at  $m$  different points

## Divide and Conquer FFT

---

- $V$  can be evaluated using divide-and-conquer

$$\begin{aligned} V(x) &= \sum_{k=0}^{m-1} x^k * v(k) \\ &= v[0] + x^2 * v[2] + x^4 * v[4] + \dots \\ &\quad + x^*(v[1] + x^2 * v[3] + x^4 * v[5] + \dots) \\ &= V_{\text{even}}(x^2) + x^* V_{\text{odd}}(x^2) \end{aligned}$$

- $V$  has degree  $m-1$ , so  $V_{\text{even}}$  and  $V_{\text{odd}}$  are polynomials of degree  $m/2-1$
- We evaluate these at  $m$  points:  $(\omega^j)^2$  for  $0 \leq j \leq m-1$
- But this is really just  $m/2$  different points, since  $(\omega^{(j+m/2)})^2 = (\omega^j * \omega^{m/2})^2 = \omega^{2j} * \omega^m = (\omega^j)^2$
- So FFT on  $m$  points reduced to 2 FFTs on  $m/2$  points
  - Divide and conquer!

## Divide-and-Conquer FFT (D&C FFT)

---

$\text{FFT}(v, \omega, m)$  ... assume  $m$  is a power of 2

if  $m = 1$  return  $v[0]$

else

$v_{\text{even}} = \text{FFT}(v[0:2:m-2], \omega^2, m/2)$

$v_{\text{odd}} = \text{FFT}(v[1:2:m-1], \omega^2, m/2)$  precomputed

$\omega\text{-vec} = [\omega^0, \omega^1, \dots \omega^{(m/2-1)}]$

return  $[v_{\text{even}} + (\omega\text{-vec} .* v_{\text{odd}}),$

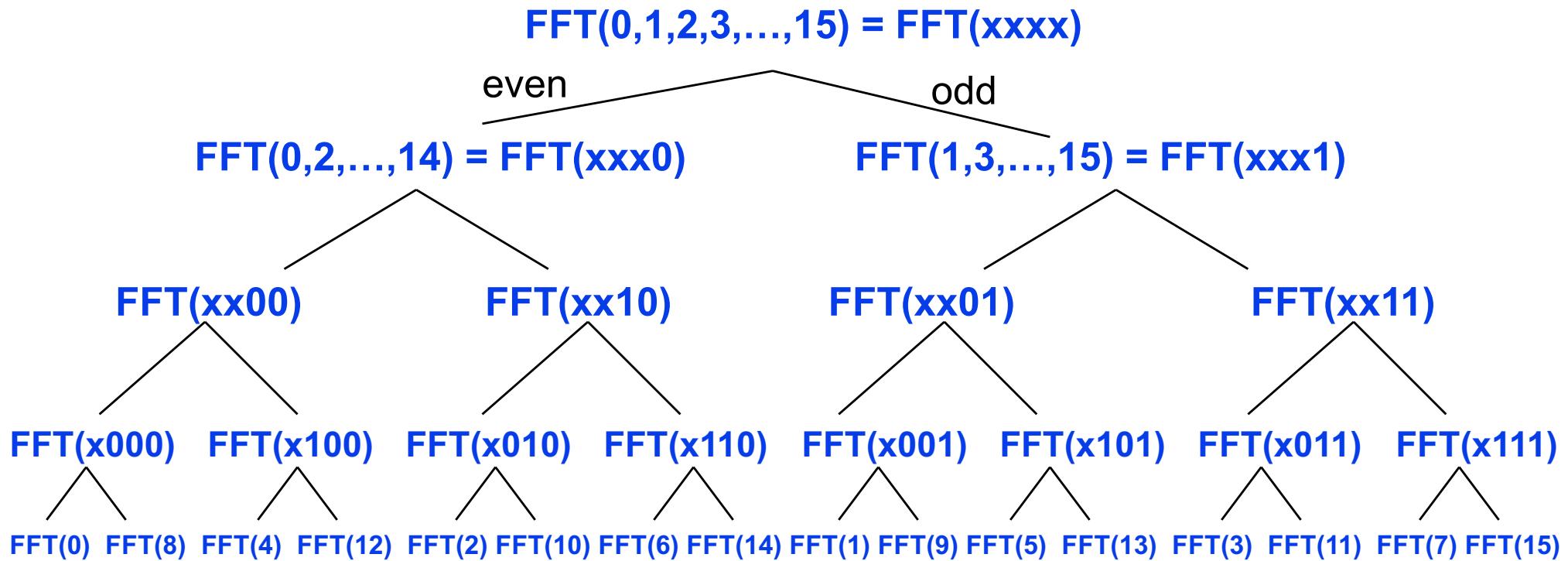
$v_{\text{even}} - (\omega\text{-vec} .* v_{\text{odd}})]$

◦ Matlab notation: “ $.$ ” means component-wise multiply.

**Cost:  $T(m) = 2T(m/2)+O(m) = O(m \log m)$  operations.**

# An Iterative Algorithm

- The call tree of the D&C FFT algorithm is a complete binary tree of  $\log m$  levels



- An **iterative** algorithm that uses loops rather than **recursion**, does each level in the tree starting at the bottom
  - Algorithm overwrites  $v[i]$  by  $(F^*v)[\text{bitreverse}(i)]$
- Practical algorithms combine recursion (for memory hierarchy) and iteration (to avoid function call overhead) – more later

## Bit Reversal

---

- ° The outputs of the FFT stored in “bit reversed” order

Decimal	Binary	Decimal	Binary
0	0000	0	0000
1	0001	8	1000
2	0010	4	0100
3	0011	12	1100
4	0100	2	0010
5	0101	10	1010
6	0110	6	0100
7	0111	14	1110
8	1000	1	0001
9	1001	9	1001
10	1010	5	0101
11	1011	13	1101
12	1100	3	0011
13	1101	11	1011
14	1110	7	0111
15	1111	15	1111

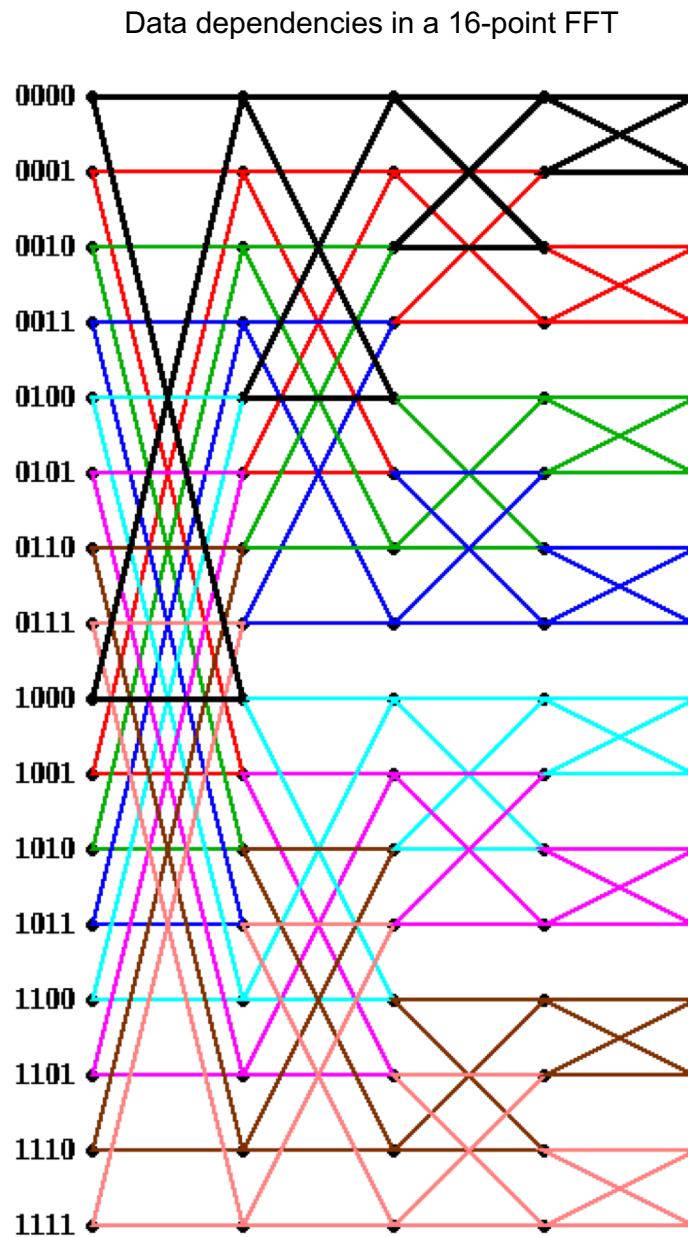


- ° May need to sort them (which is like transposes at various levels)
- ° Although many FFT uses, will do a reverse FFT as well, and therefore not require a bit reversal at all

# Parallel 1D FFT

---

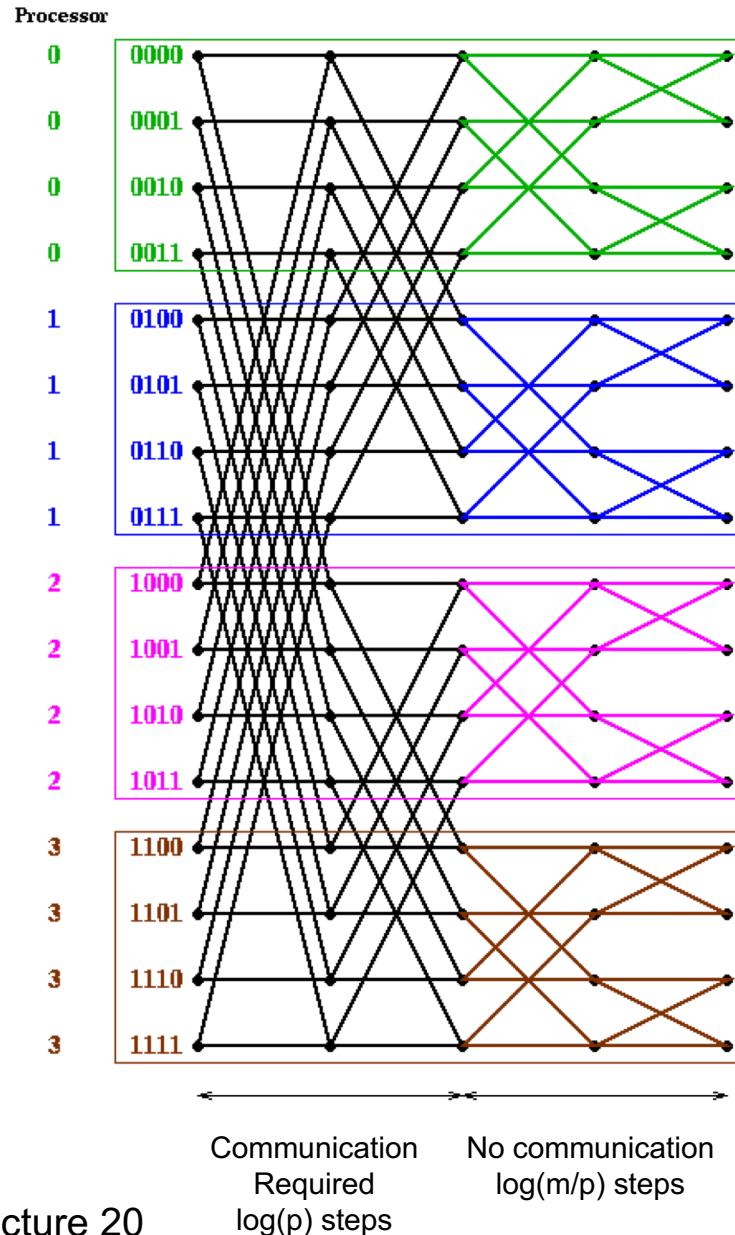
- **Data dependencies in 1D FFT**
  - Butterfly pattern
  - From  $v_{\text{even}} \pm w \cdot^* v_{\text{odd}}$
- **A PRAM algorithm takes  $O(\log m)$  time**
  - each step to right is parallel
  - there are  $\log m$  steps
- **What about communication cost?**
- **(See UCB EECS Tech report UCB/CSD-92-713 for details, aka “LogP paper”)**



# Block Layout of 1D FFT

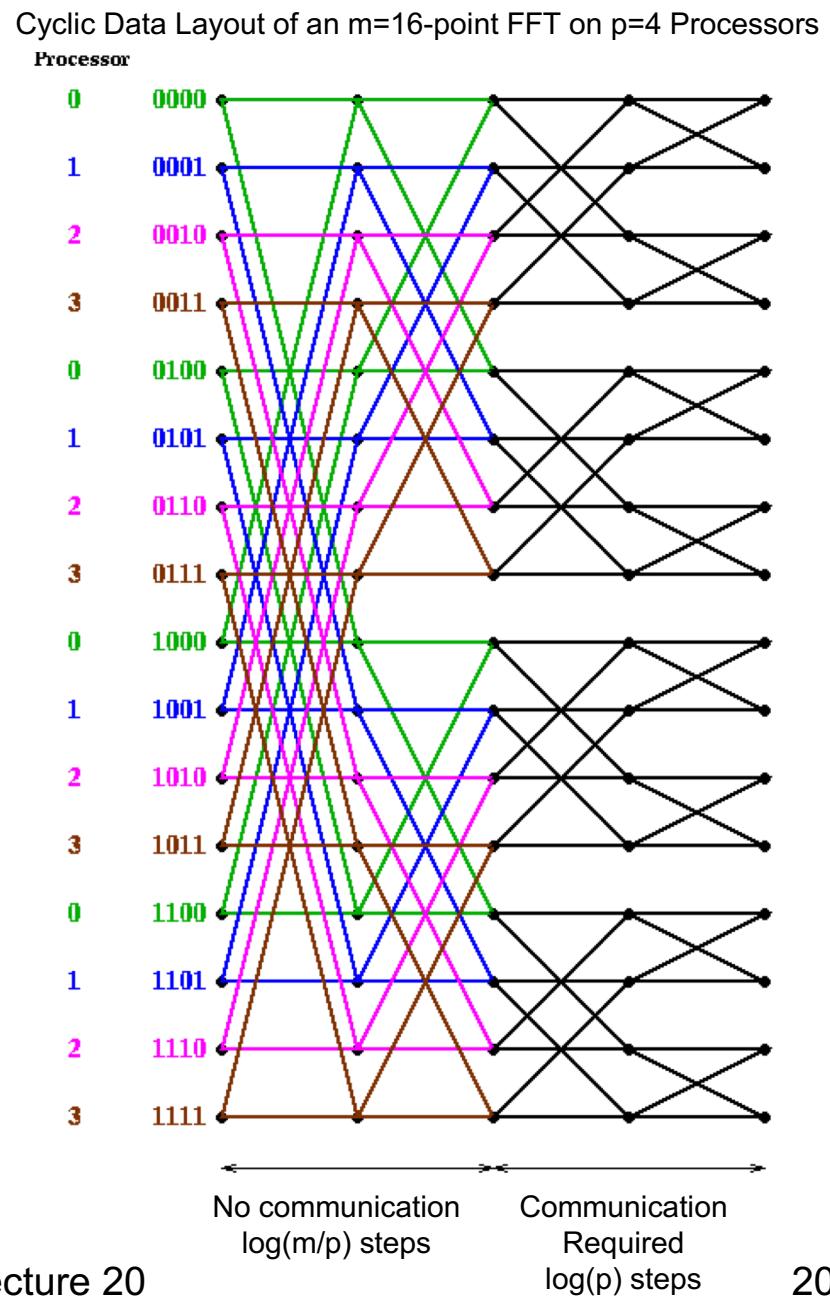
- **Using a block layout (m/p contiguous words per processor)**
- **No communication in last  $\log m/p$  steps**
- **Significant communication in first  $\log p$  steps**

Block Data Layout of an  $m=16$ -point FFT on  $p=4$  Processors



# Cyclic Layout of 1D FFT

- **Cyclic layout  
(consecutive words map to consecutive processors)**
- **No communication in first  $\log(m/p)$  steps**
- **Communication in last  $\log(p)$  steps**



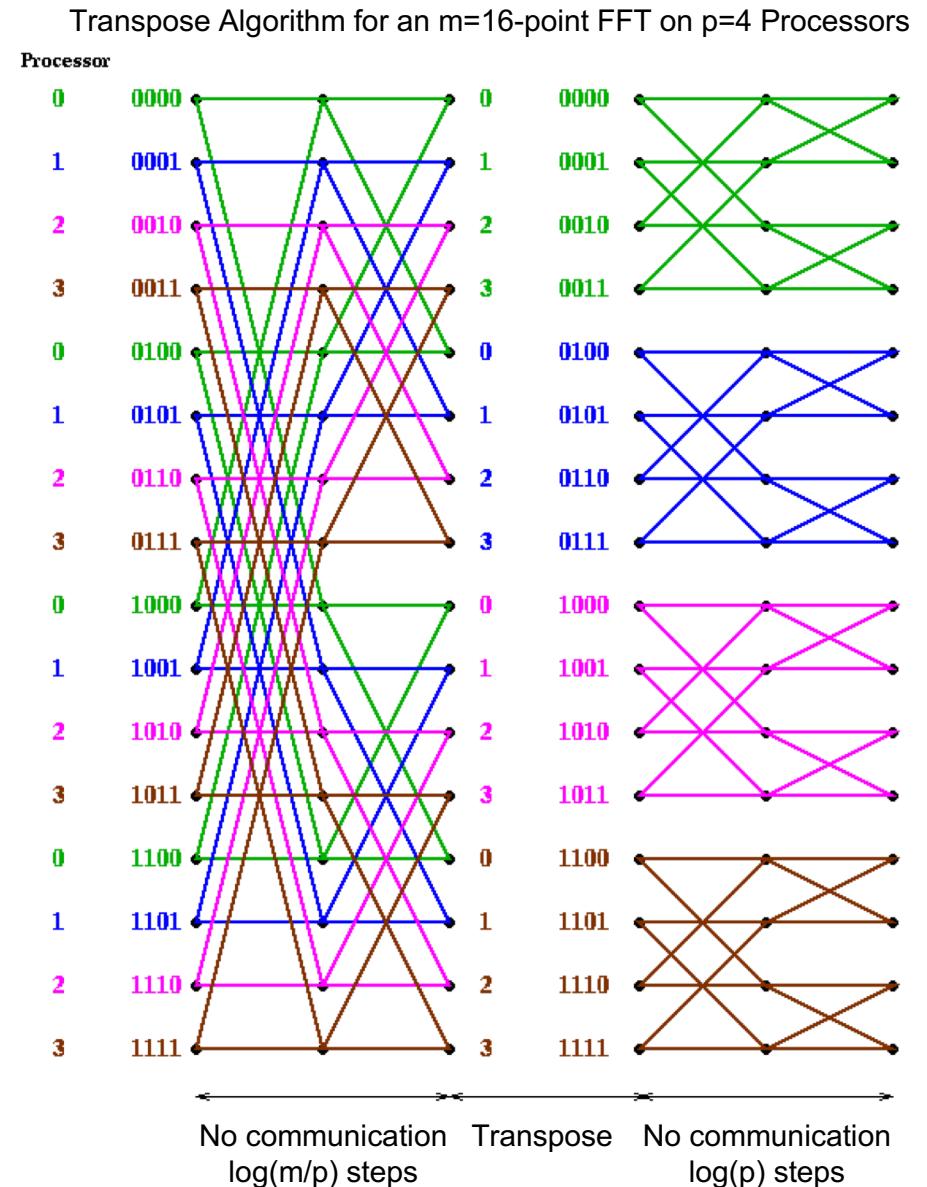
## Parallel Complexity

---

- $m = \text{vector size}$ ,  $p = \text{number of processors}$
- $f = \text{time per flop} = 1$
- $\alpha = \text{latency for message}$
- $\beta = \text{time per word in a message}$
- $\text{Time(block\_FFT)} = \text{Time(cyclic\_FFT)} =$ 
  - $2*m*\log(m)/p$  ... perfectly parallel flops
  - $+ \log(p) * \alpha$  ... 1 message/stage,  $\log p$  stages
  - $+ m*\log(p)/p * \beta$  ...  $m/p$  words/message

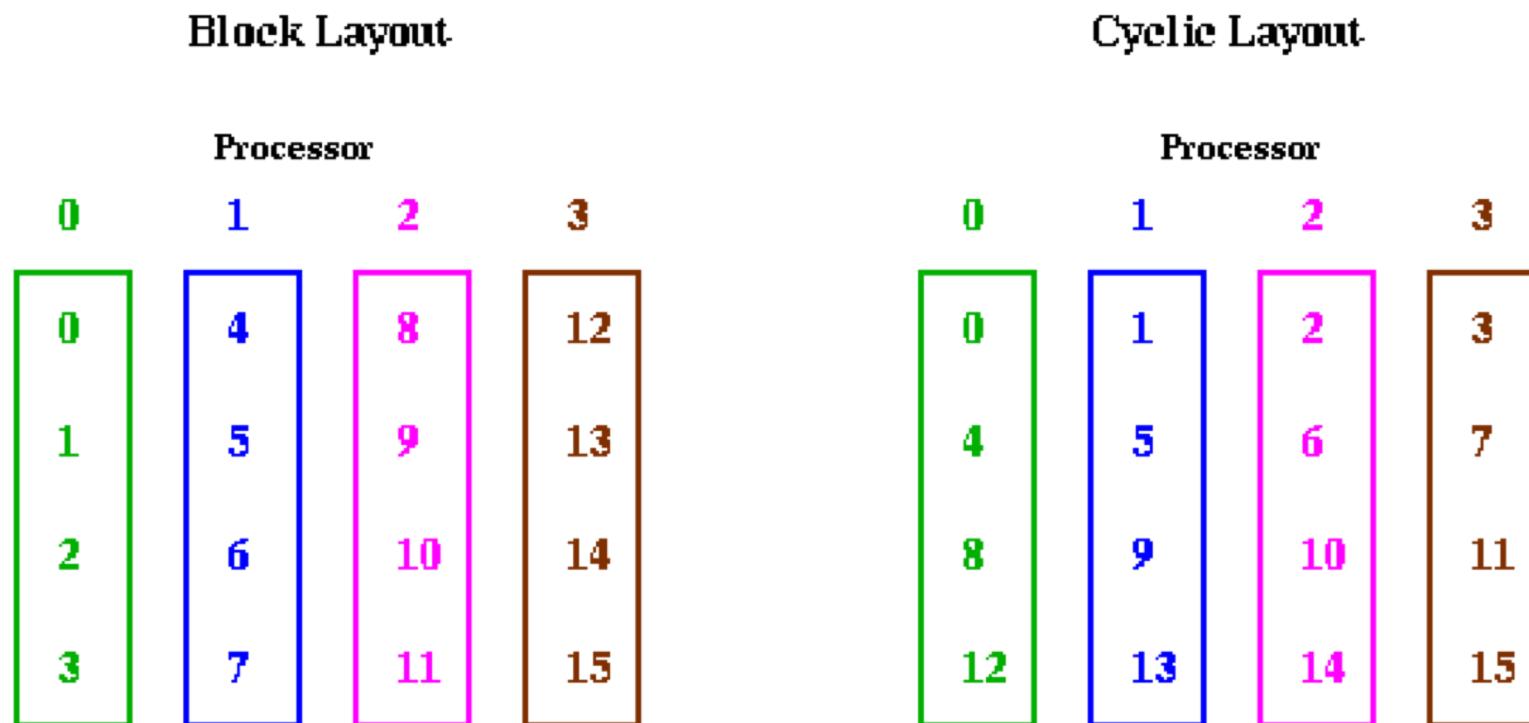
# FFT With “Transpose”

- If we start with a cyclic layout for first  $\log(m/p)$  steps, there is no communication
- Then transpose the vector for last  $\log(p)$  steps
- All communication is in the transpose
- Note: This example has  $\log(m/p) = \log(p)$ 
  - If  $\log(m/p) < \log(p)$  more phases/layouts will be needed
  - We will assume  $\log(m/p) \geq \log(p)$  for simplicity, i.e.  $m \geq p^2$



# Why is the Communication Step Called a Transpose?

- ° Analogous to transposing an array
- ° View as a 2D array of  $m/p$  by  $p$
- ° Note: same idea is useful for caches



## Parallel Complexity of the FFT with Transpose

---

- ° If no communication is pipelined (overestimate!)
- ° Time(transposeFFT) =

$2*m*log(m)/p$	same as before
$+ (p-1) * \alpha$	was $\log(p) * \alpha$
$+ m*(p-1)/p^2 * \beta$	was $m * \log(p)/p * \beta$
- ° If communication is pipelined, so we do not pay for  $p-1$  messages, the second term becomes simply  $\alpha$ , rather than  $(p-1)\alpha$
- ° This is close to optimal. See LogP paper for details.
- ° See also following papers
  - A. Sahai, “Hiding Communication Costs in Bandwidth Limited FFT”
  - R. Nishtala et al, “Optimizing bandwidth limited problems using one-sided communication”

# Sequential Communication Complexity of the FFT

---

- How many words need to be moved between main memory and cache of size  $M$  to do the FFT of size  $m$ , where  $m > M$ ?
- Thm (Hong, Kung, 1981): #words =  $\Omega(m \log m / \log M)$ 
  - Proof follows from each word of data being reusable only  $\log M$  times; assumes no recomputation
- Attained by transpose algorithm
  - Sequential algorithm “simulates” parallel algorithm
  - Imagine we have  $P = m/M$  processors, so each processor stores and works on  $O(M)$  words
  - Each local computation phase in parallel FFT replaced by similar phase working on cache resident data in sequential FFT
  - Each communication phase in parallel FFT replaced by reading/writing data from/to cache in sequential FFT
- Attained by recursive, “cache-oblivious” algorithm (FFTW)

# Parallel Communication Complexity of the FFT

---

- How many words need to be moved between  $p$  processors to do the FFT of size  $m$ ?
- Thm (Aggarwal, Chandra, Snir, 1990): #words =  $\Omega(m \log m / (p \log m/p))$ 
  - Proof assumes no recomputation
  - Holds independent of local memory size (which must exceed  $m/p$ )
- Does TransposeFFT attain lower bound?
  - Recall assumption:  $\log(m/p) \geq \log(p)$
  - So  $2 \geq \log(m) / \log(m/p) \geq 1$
  - So #words =  $\Omega(m / p)$
  - Attained by transpose algorithm

## Comment on the 1D Parallel FFT

---

- ° **The above algorithm leaves data in bit-reversed order**
  - Some applications can use it this way, like Poisson
  - Others require another transpose-like operation
- ° **Other parallel algorithms also exist**
  - A very different 1D FFT is due to Edelman
    - <http://www-math.mit.edu/~edelman>
  - Based on the Fast Multipole algorithm
  - Less communication for non-bit-reversed algorithm
  - Approximates FFT

## Higher Dimensional FFTs

---

- ° **FFTs on 2 or more dimensions are defined as 1D FFTs on vectors in all dimensions.**
  - 2D FFT does 1D FFTs on all rows and then all columns
- ° **There are 3 obvious possibilities for the 2D FFT:**
  - (1) 2D blocked layout for matrix, using parallel 1D FFTs for each row and column
  - (2) Block row layout for matrix, using serial 1D FFTs on rows, followed by a transpose, then more serial 1D FFTs
  - (3) Block row layout for matrix, using serial 1D FFTs on rows, followed by parallel 1D FFTs on columns
  - Option 2 is best, if we overlap communication and computation
- ° **For a 3D FFT the options are similar**
  - 2 phases done with serial FFTs, followed by a transpose for 3rd
  - can overlap communication with 2nd phase in practice

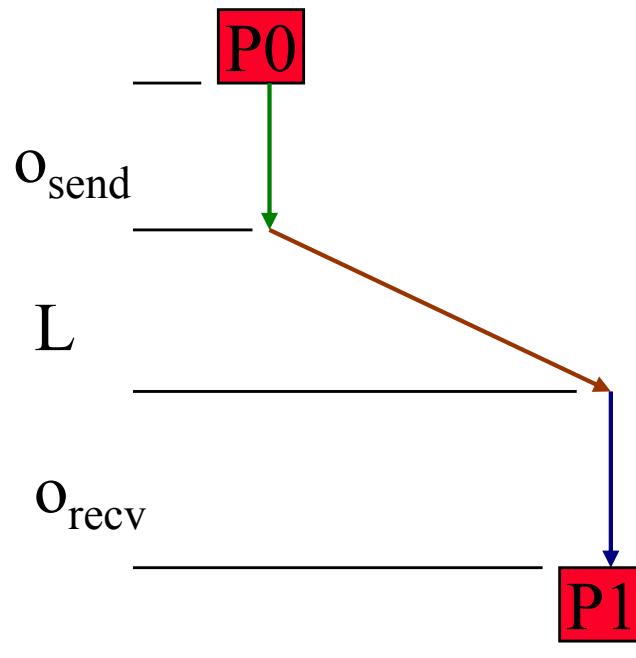
## Bisection Bandwidth

---

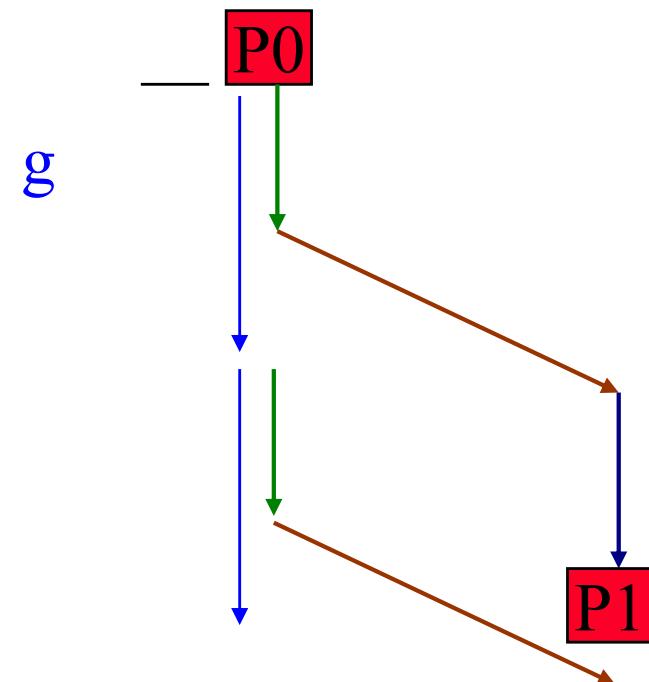
- **FFT requires one (or more) transpose operations:**
  - Every processor sends  $1/p$ -th of its data to each other one
- **Bisection Bandwidth limits this performance**
  - Bisection bandwidth is the bandwidth across the narrowest part of the network
  - Important in global transpose operations, all-to-all, etc.
- **“Full bisection bandwidth” is expensive**
  - Fraction of machine cost in the network is increasing
  - Fat-tree and full crossbar topologies may be too expensive
  - Especially on machines with 100K and more processors
  - SMP clusters often limit bandwidth at the node level
- **Goal: overlap communication and computation**

## Modified LogGP Model

◦ LogGP: no overlap



◦ LogGP: with overlap

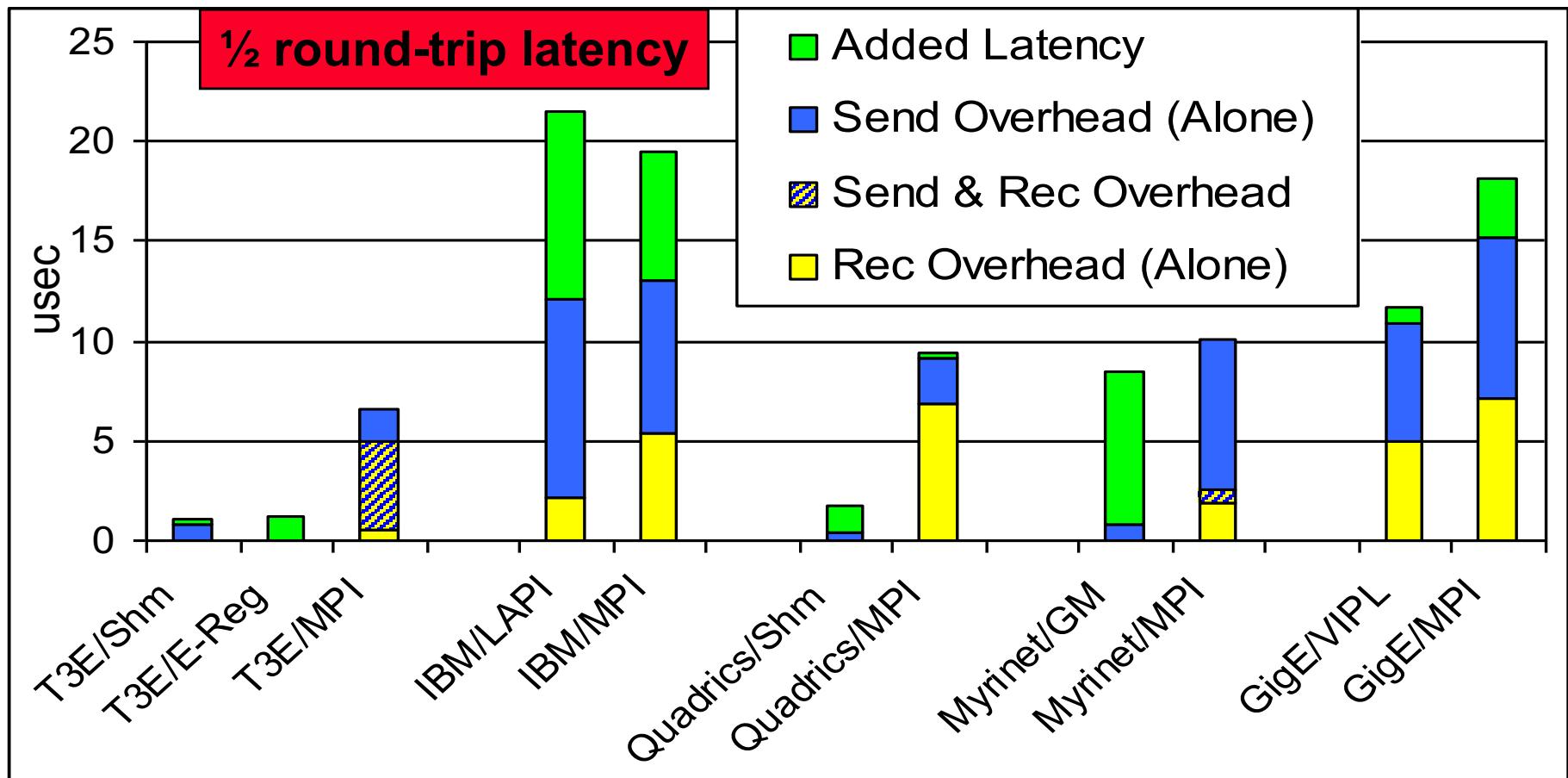


EEL: end to end latency (1/2 roundtrip)

$g$ : minimum time between small message sends

$G$ : gap per byte for larger messages

# Historical Perspective



- Potential performance advantage for fine-grained, one-sided programs
- Potential productivity advantage for irregular applications

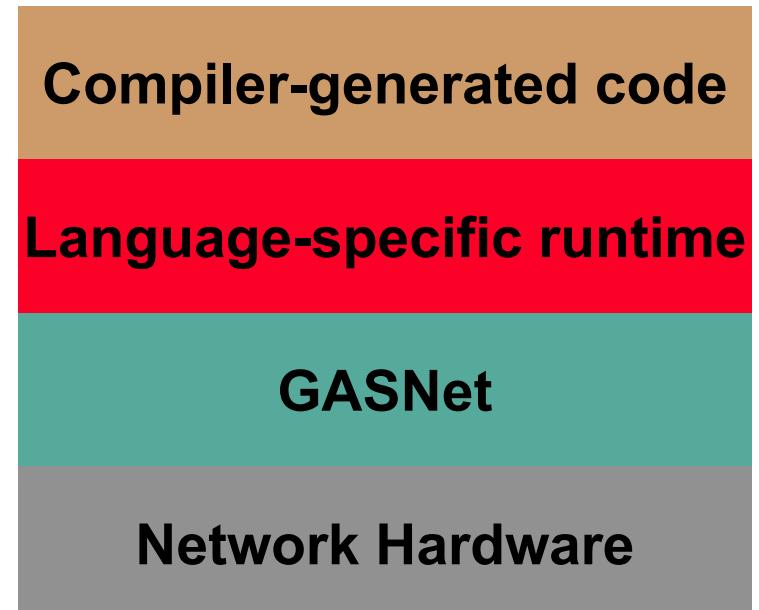
## General Observations

---

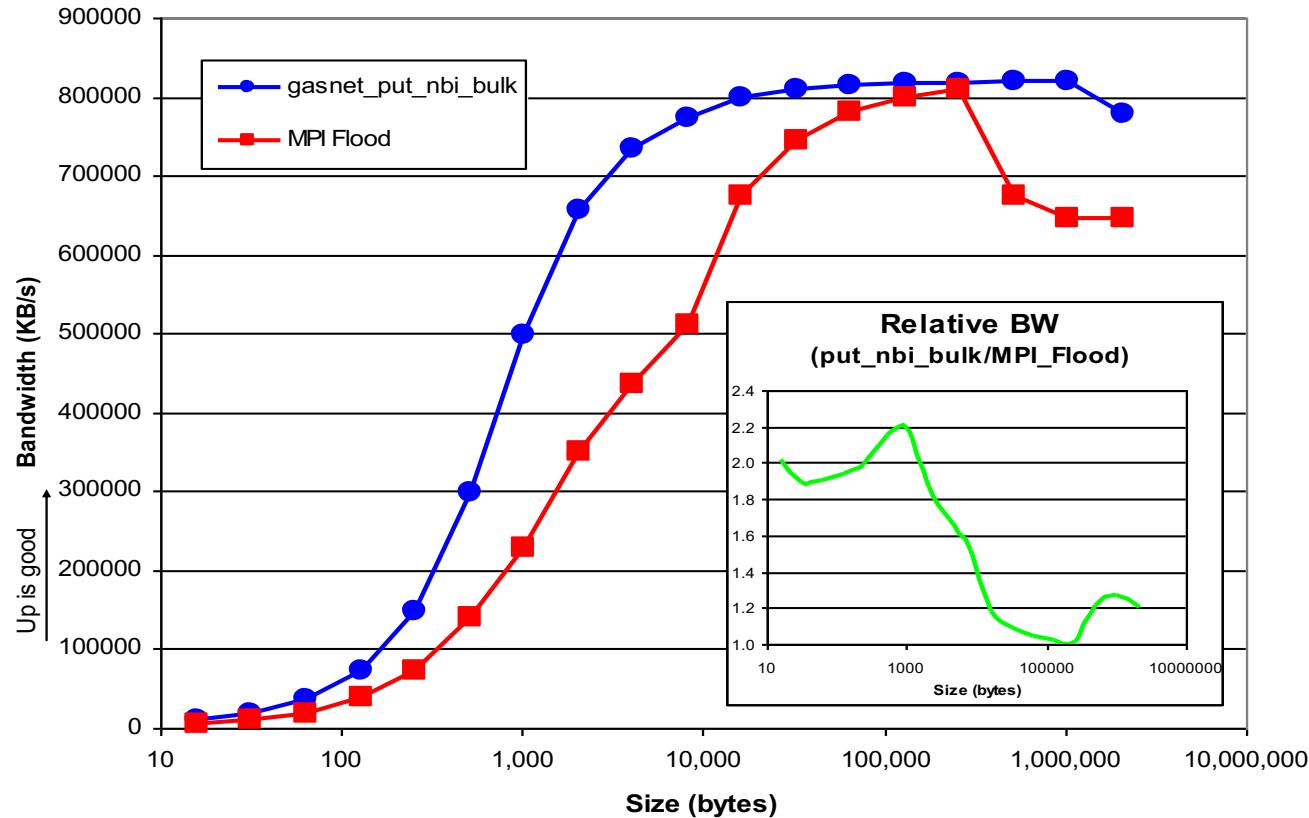
- “Overlap” means computing and communicating simultaneously, (or communication with other communication, aka pipelining)
- Rest of slide about comm/comp overlap
- The overlap potential is the difference between the gap and overhead
  - No potential if CPU is tied up throughout message send
    - E.g., no send-side DMA
  - Potential grows with message size for machines with DMA (per byte cost is handled by network, i.e. NIC)
  - Potential grows with amount of network congestion
    - Because gap grows as network becomes saturated
- Remote overhead is 0 for machine with RDMA
- Need good SW support to take advantage of this

### GASNet offers put/get communication

- One-sided: no remote CPU involvement required in API  
**(key difference with MPI)**
  - Message contains remote address
  - No need to match with a receive
  - No implicit ordering required
- Used in language runtimes (UPC, etc.)
- Fine-grained and bulk transfers
- Split-phase communication

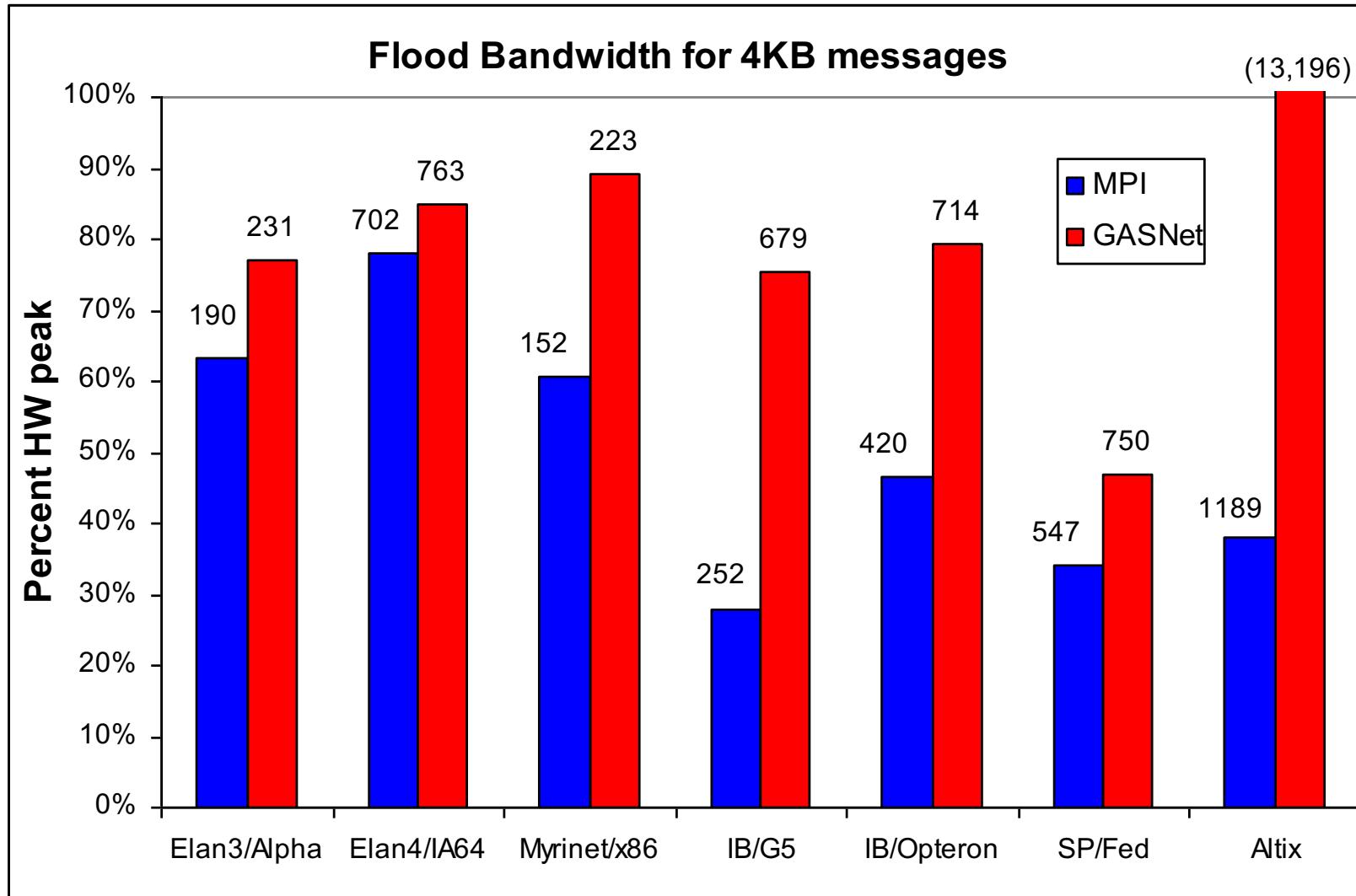


# Performance of 1-Sided vs 2-sided Communication: GASNet vs MPI



- Comparison on Opteron/InfiniBand – GASNet’s vapi-conduit and OSU MPI 0.9.5
- Up to large message size (> 256 Kb), GASNet provides up to 2.2X improvement in streaming bandwidth
- Half power point ( $N/2$ ) differs by one order of magnitude

## GASNet: Performance for mid-range message sizes



GASNet usually reaches saturation bandwidth before MPI - fewer costs to amortize  
Usually outperforms MPI at medium message sizes - often by a large margin

# NAS FT Benchmark Case Study

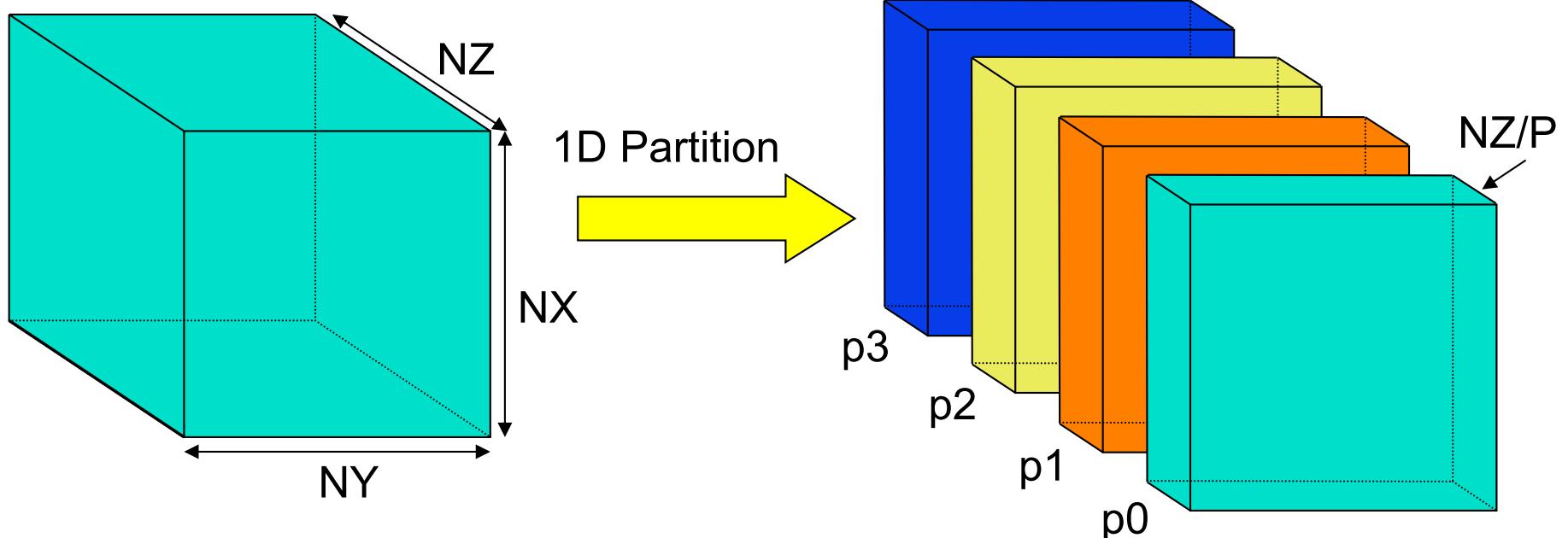
---

- **Performance of Exchange (All-to-all) is critical**
  - Communication to computation ratio increases with faster, more optimized 1-D FFTs (used best available, from FFTW)
  - Determined by available bisection bandwidth
  - Between 30-40% of the application's total runtime
- **Assumptions**
  - 1D partition of 3D grid
  - At most N processors for  $N^3$  grid
  - HPC Challenge benchmark has large 1D FFT (can be viewed as 3D or more with proper roots of unity)
- **Reference for details**
  - “Optimizing Bandwidth Limited Problems Using One-side Communication and Overlap”, C. Bell, D. Bonachea, R. Nishtala, K. Yelick, IPDPS’ 06 ([www.eecs.berkeley.edu/~rajashn](http://www.eecs.berkeley.edu/~rajashn))
  - Started as CS267 project

## Performing a 3D FFT (1/3)

- ° **NX x NY x NZ elements spread across P processors**
- ° **Will Use 1-Dimensional Layout in Z dimension**
  - Each processor gets  $NZ / P$  “planes” of  $NX \times NY$  elements per plane

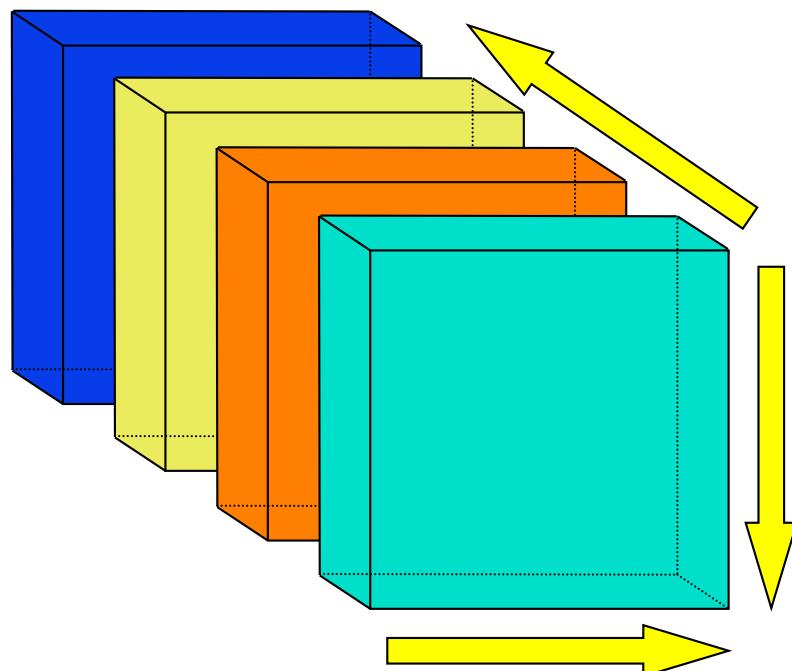
Example:  $P = 4$



## Performing a 3D FFT (2/3)

---

- ° Perform an FFT in all three dimensions
- ° With 1D layout, 2 out of the 3 dimensions are local while the last Z dimension is distributed



Step 1: FFTs on the columns  
(all elements local)

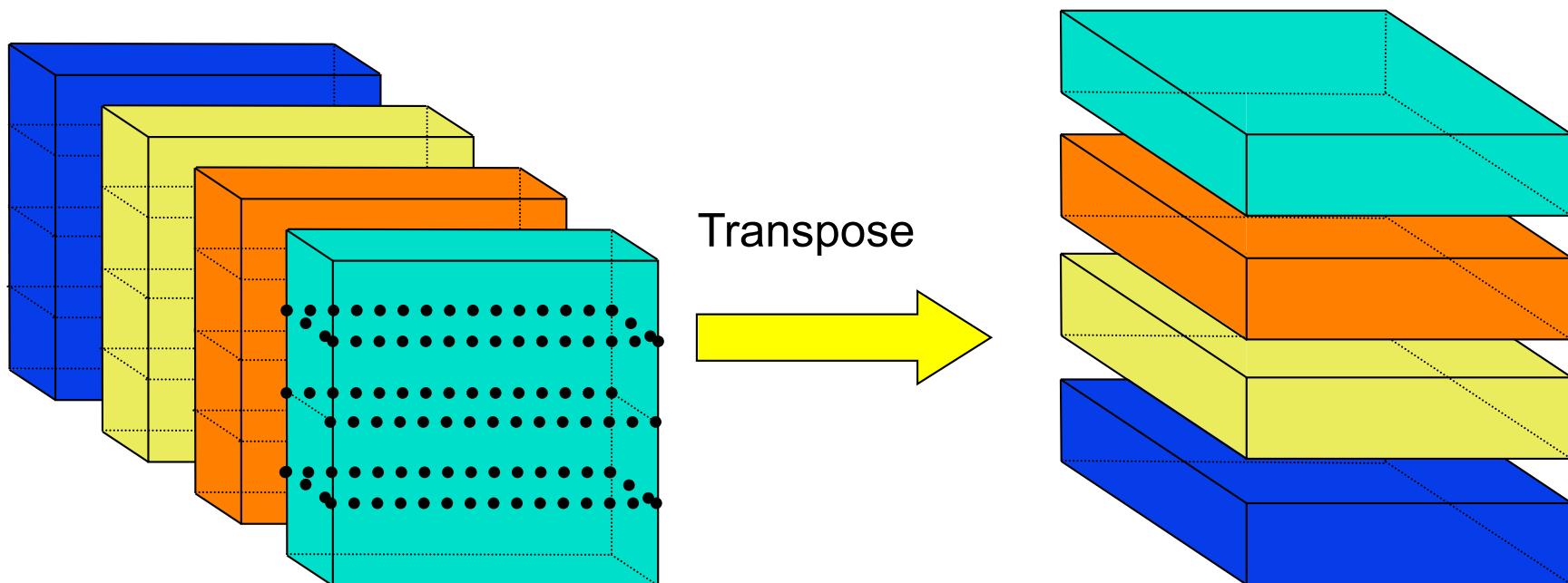
Step 2: FFTs on the rows  
(all elements local)

Step 3: FFTs in the Z-dimension  
(requires communication)

## Performing the 3D FFT (3/3)

---

- ° Can perform Steps 1 and 2 since all the data is available without communication
- ° Perform a Global Transpose of the cube
  - Allows step 3 to continue



# The Transpose

---

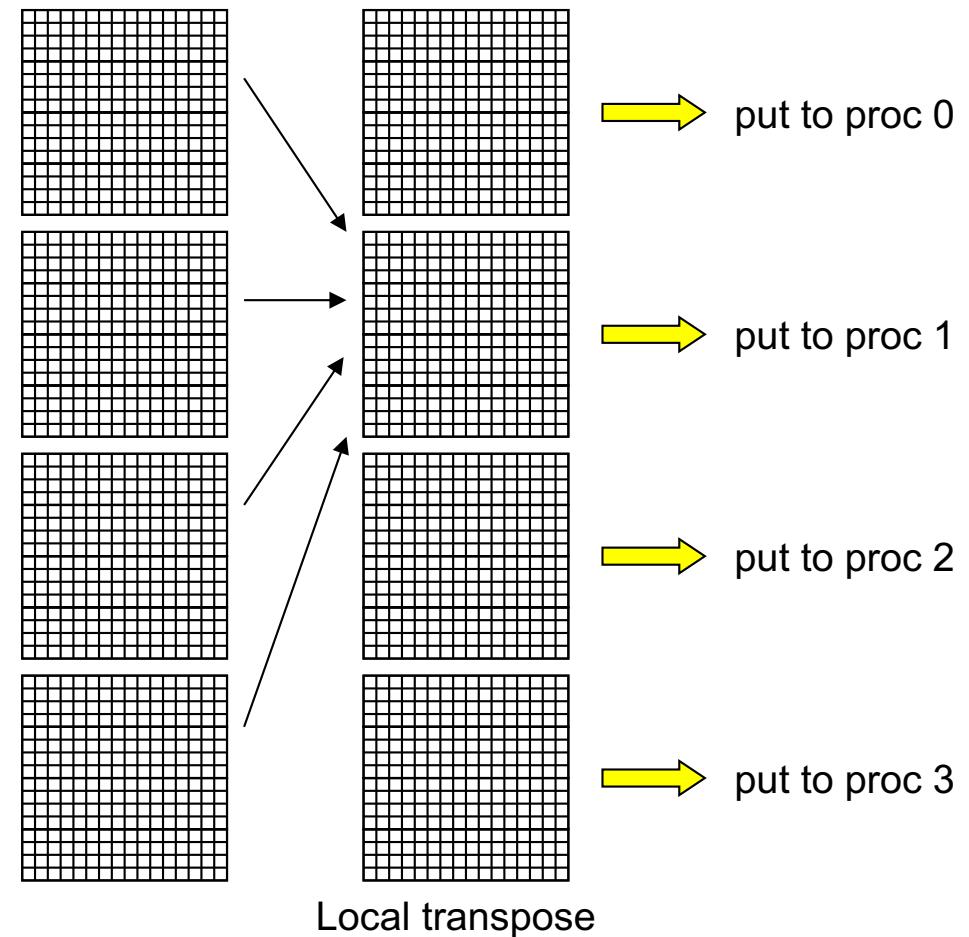
- Each processor has to scatter input domain to other processors
  - Every processor divides its portion of the domain into P pieces
  - Send each of the P pieces to a different processor
- Three different ways to break up the messages
  1. Packed Slabs (i.e. single packed “All-to-all” in MPI parlance) (3D)
  2. Slabs (2D)
  3. Pencils (1D)
- Going from approach Packed Slabs to Slabs to Pencils leads to
  - An order of magnitude **increase in the number of messages**
  - An order of magnitude **decrease in the size of each message**
- Why do this? Slabs and Pencils allow overlapping communication and computation and leverage RDMA support in modern networks

## Algorithm 1: Packed Slabs

---

Example with  $P=4$ ,  $NX=NY=NZ=16$

1. Perform all row and column FFTs
2. Perform local transpose
  - data destined to a remote processor are grouped together
3. Perform  $P$  puts of the data



- For  $512^3$  grid across 64 processors
  - Send 64-1 messages of 512kB each

Source: R. Nishtala, C. Bell, D. Bonachea, K. Yelick

## Bandwidth Utilization

---

- ° NAS FT (Class D) with 256 processors on Opteron/InfiniBand
  - Each processor sends 256-1 messages of 512kBytes
  - Global Transpose (i.e. all to all exchange) only achieves 67% of peak point-to-point bidirectional bandwidth
  - Many factors could cause this slowdown
    - Network contention
    - Number of processors with which each processor communicates
- ° Can we do better?

## Algorithm 2: Slabs

- ° Waiting to send all data in one phase bunches up communication events

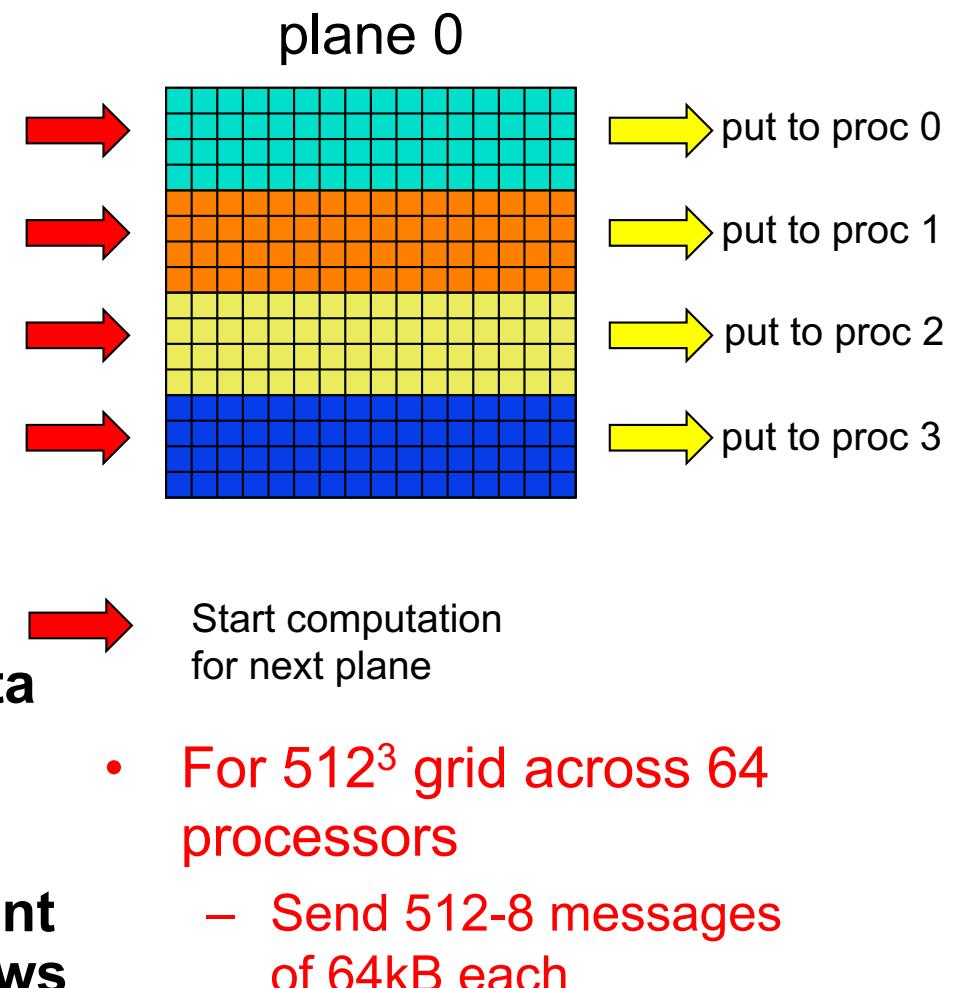
- ° Algorithm Sketch

- for each of the  $NZ/P$  planes
    - Perform all column FFTs
    - for each of the  $P$  “slabs”  
(a slab is  $NX/P$  rows)
      - Perform FFTs on the rows in the slab
      - Initiate 1-sided put of the slab

- Wait for all puts to finish
  - Barrier

- ° Non-blocking RDMA puts allow data movement to be overlapped with computation.

- ° Puts are spaced apart by the amount of time to perform FFTs on  $NX/P$  rows



## Algorithm 3: Pencils

- ° Further reduce the granularity of communication

- Send a row (*pencil*) as soon as it is ready

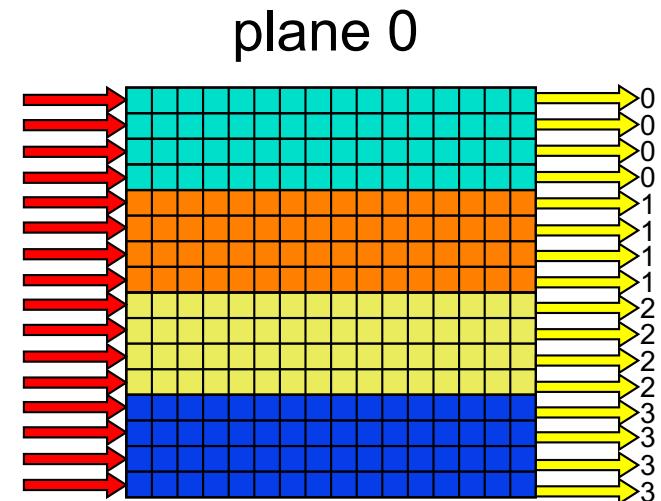
- ° Algorithm Sketch

- For each of the NZ/P planes
    - Perform all 16 column FFTs
    - For  $r=0; r < NX/P; r++$ 
      - For each slab s in the plane
        - Perform FFT on row r of slab s
        - Initiate 1-sided put of row r
    - Wait for all puts to finish
    - Barrier

- ° Large increase in message count

- ° Communication events finely diffused through computation

- Maximum amount of overlap
  - Communication starts early



→ Start computation  
for next plane

- For  $512^3$  grid across 64 processors
  - Send 4096-64 messages of 8kB each

Source: R. Nishtala, C. Bell, D. Bonachea, K. Yelick

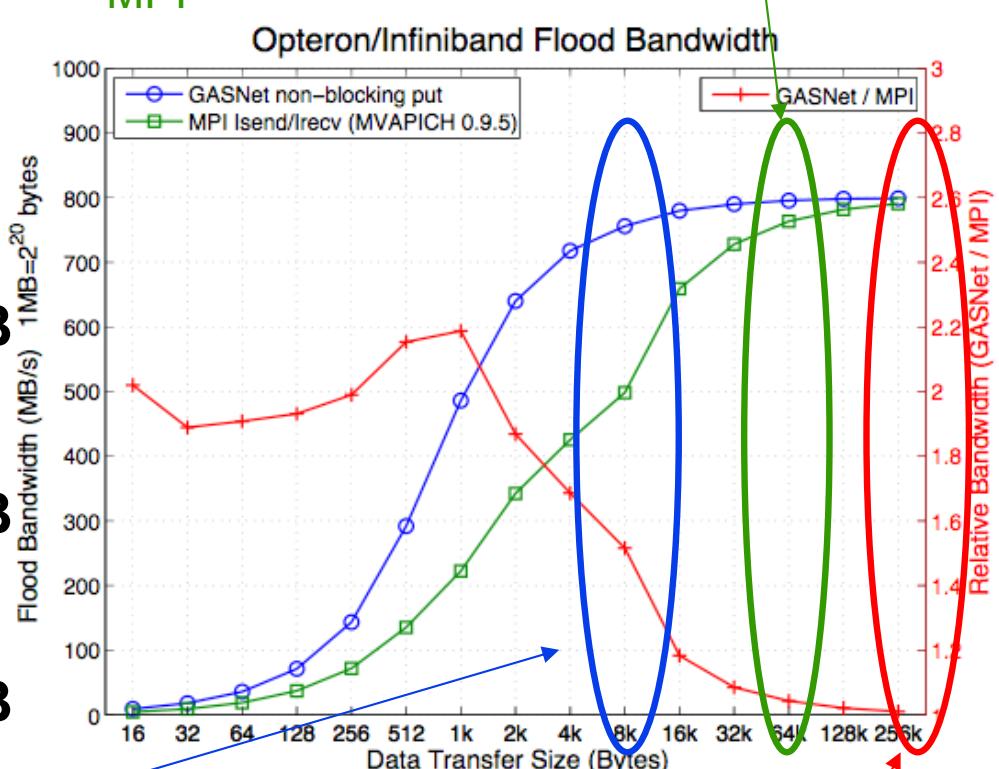
# Communication Requirements

- ° **512<sup>3</sup> across 64 processors**

- **Alg 1: Packed Slabs**
  - Send 64 messages of 512kB
- **Alg 2: Slabs**
  - Send 512 messages of 64kB
- **Alg 3: Pencils**
  - Send 4096 messages of 8kB

GASNet achieves close to peak bandwidth with Pencils but MPI is about 50% less efficient at 8k  
More overlap possible with 8k messages

With Slabs GASNet is slightly faster than MPI



With the message sizes in Packed Slabs both comm systems reach the same peak bandwidth

# Platforms

---

Name	Processor	Network	Software
Opteron/Infiniband “Jacquard” @ NERSC	Dual 2.2 GHz Opteron (320 nodes @ 4GB/node)	Mellanox Cougar InfiniBand 4x HCA	Linux 2.6.5, Mellanox VAPI, MVAPICH 0.9.5, Pathscale CC/F77 2.0
Alpha/Elan3 “Lemieux” @ PSC	Quad 1 GHz Alpha 21264 (750 nodes @ 4GB/node)	Quadrics QsNet1 Elan3 /w dual rail (one rail used)	Tru64 v5.1, Elan3 libelan 1.4.20, Compaq C V6.5-303, HP Fortra Compiler X5.5A-4085-48E1K
Itanium2/Elan4 “Thunder” @ LLNL	Quad 1.4 Ghz Itanium2 (1024 nodes @ 8GB/node)	Quadrics QsNet2 Elan4	Linux 2.4.21-chaos, Elan4 libelan 1.8.14, Intel ifort 8.1.025, icc 8.1.029
P4/Myrinet “FSN” @ UC Berkeley Millennium Cluster	Dual 3.0 Ghz Pentium 4 Xeon (64 nodes @ 3GB/node)	Myricom Myrinet 2000 M3S-PCI64B	Linux 2.6.13, GM 2.0.19, Intel ifort 8.1-20050207Z, icc 8.1-20050207Z

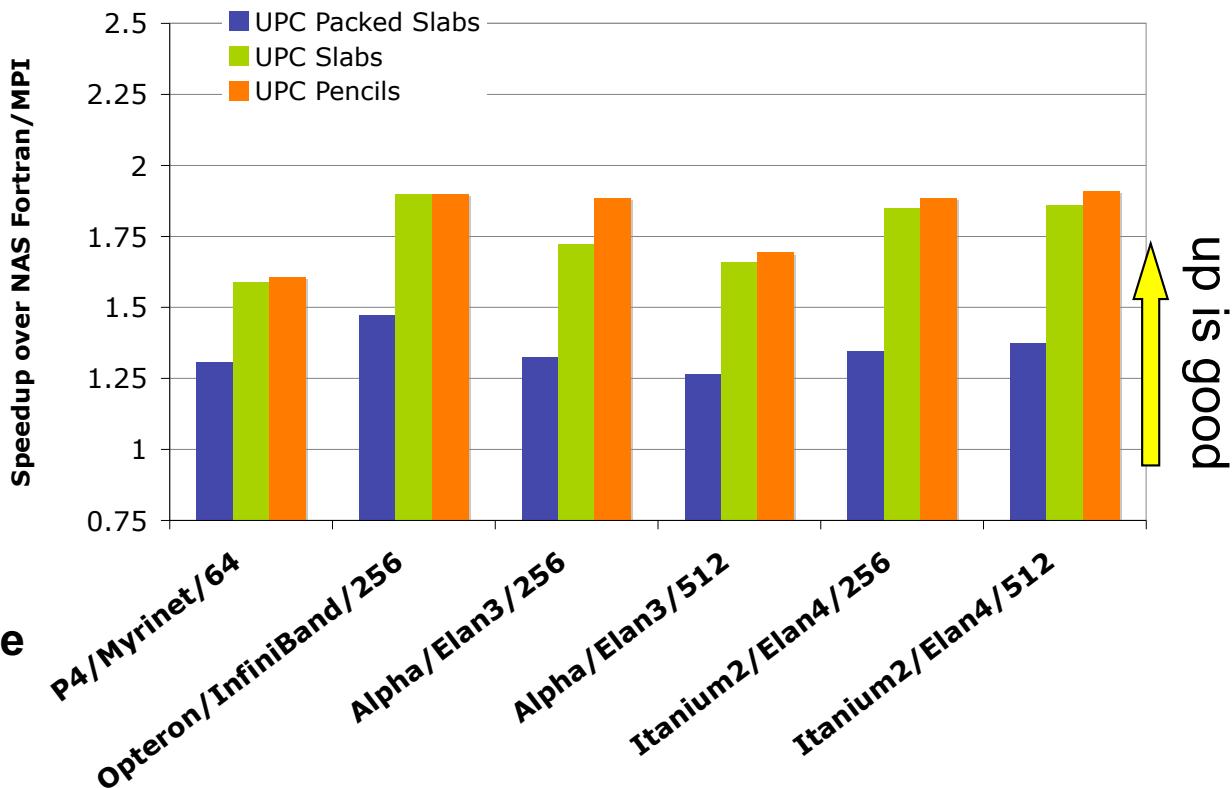
# Comparison of Algorithms

- Compare 3 algorithms against original NAS FT

- All versions including Fortran use FFTW for local 1D FFTs
- Largest class that fit in the memory (usually class D)

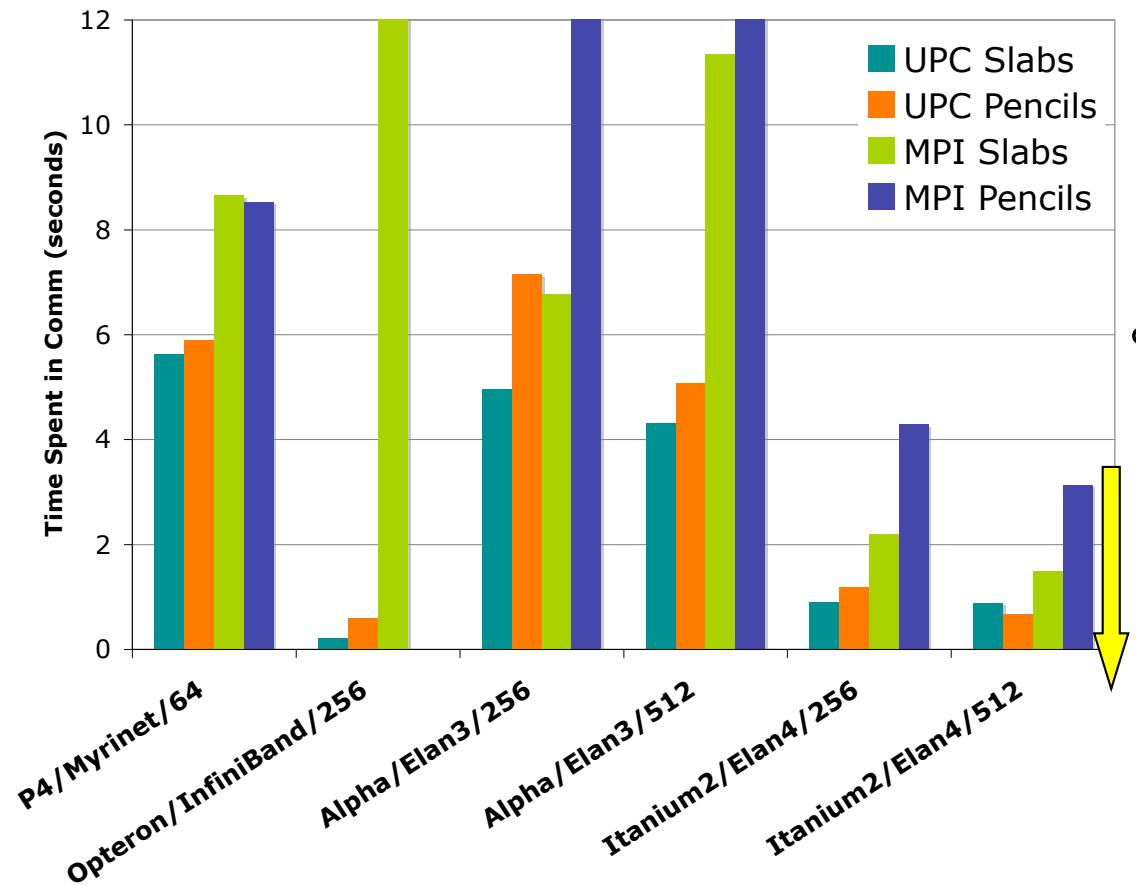
- All UPC flavors outperform original Fortran/MPI implantation by at least 20%

- One-sided semantics allow even exchange based implementations to improve over MPI implementations
- Overlap algorithms spread the messages out, easing the bottlenecks
- ~1.9x speedup in the best case



Source: R. Nishtala, C. Bell, D. Bonachea, K. Yelick

# Time Spent in Communication

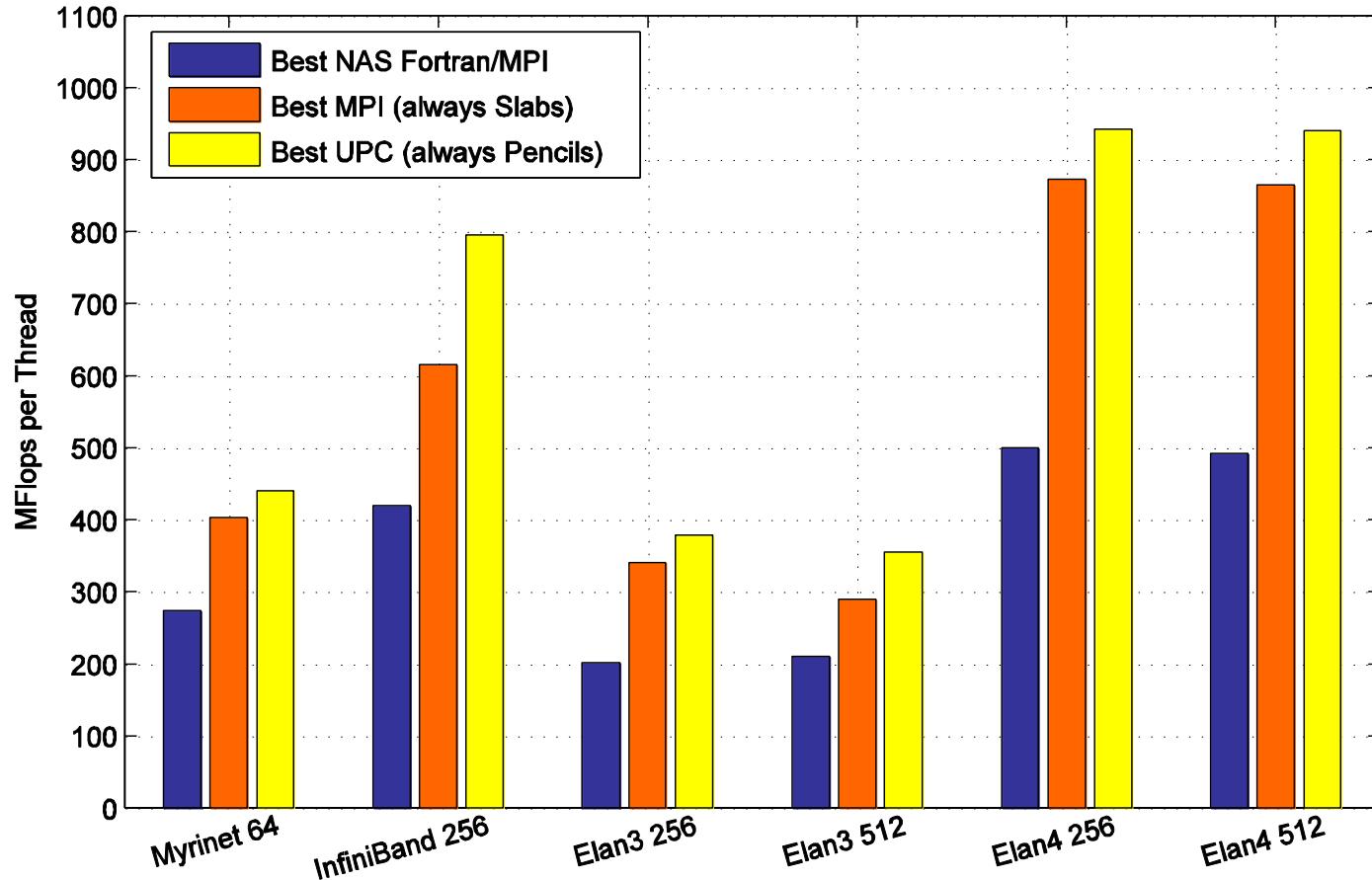


- Implemented the 3 algorithms in MPI using `Irecv`s and `Isend`s
- Compare time spent initiating or waiting for communication to finish
  - UPC consistently spends less time in communication than its MPI counterpart
  - MPI unable to handle pencils algorithm in some cases

down is good

# NAS FT Variants Performance Summary

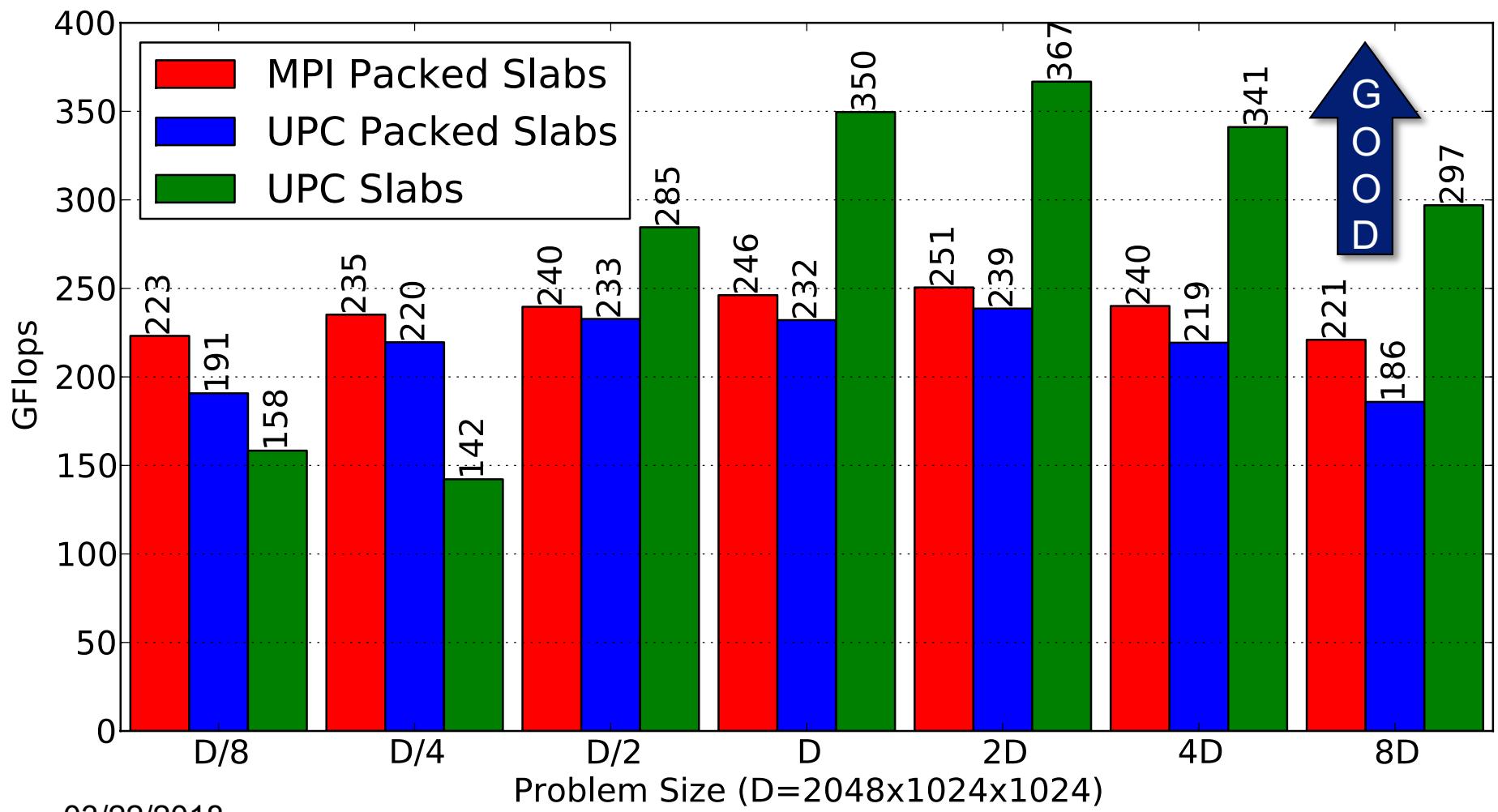
---



- ° Shown are the largest classes/configurations possible on each test machine
- ° MPI not particularly tuned for many small/medium size messages in flight (long message matching queue depths)

# FFT Performance on Cray XT4 (Franklin)

- 1024 Cores of the Cray XT4
  - Uses FFTW for local FFTs
  - Larger the problem size the more effective the overlap



# FFTW – Fastest Fourier Transform in the West

---

- [www.fftw.org](http://www.fftw.org)
- **Produces FFT implementation optimized for**
  - Your version of FFT (complex, real,...)
  - Your value of n (arbitrary, possibly prime)
  - Your architecture
  - Very good sequential performance (competes with Spiral)
- **Similar in spirit to PHIPAC/ATLAS/OSKI/Sparsity**
- **Won 1999 Wilkinson Prize for Numerical Software**
- **Widely used**
  - Latest version 3.3.7 (Oct 2017)
  - Includes threads, OpenMP, MPI versions, new architecture support
  - Layout constraints from users/apps + network differences are hard to support



the “Fastest  
Fourier Transform  
in the West”

- C library for real & complex FFTs (arbitrary size/dimensionality)  
(+ parallel versions for threads & MPI)
- Computational kernels (80% of code) automatically generated
- Self-optimizes for your hardware (picks best composition of steps)  
= portability + performance

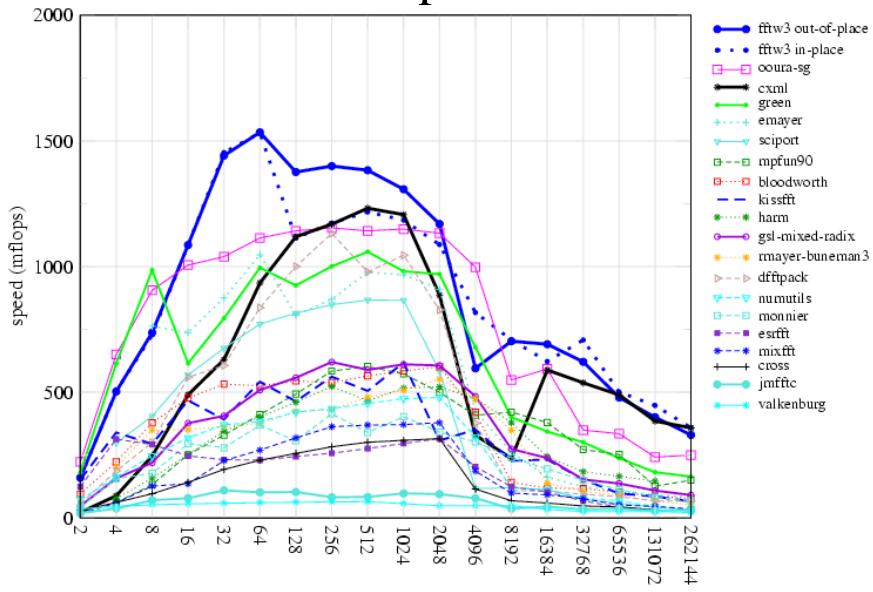
free software: <http://www.fftw.org/>



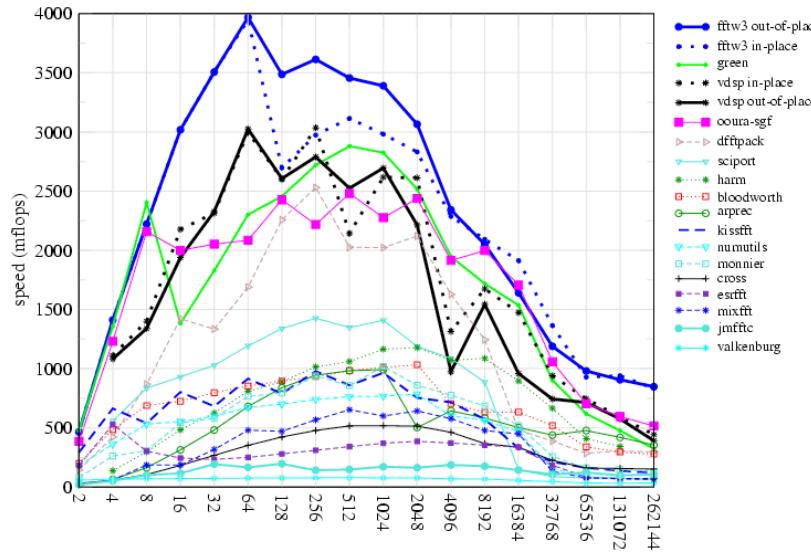
# FFTW performance

power-of-two sizes, double precision

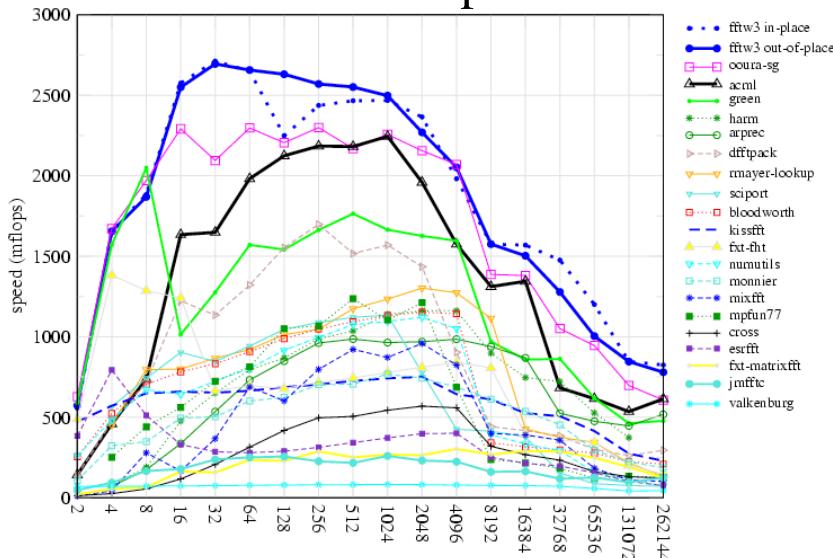
833 MHz Alpha EV6



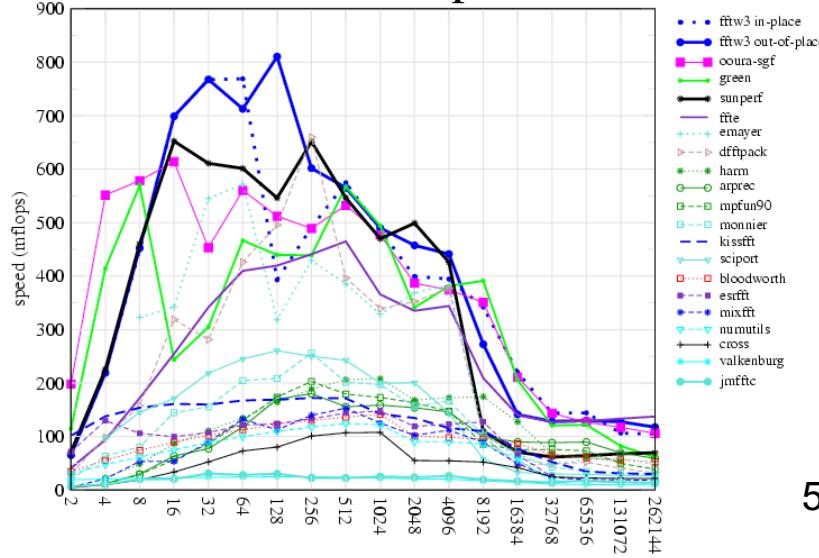
2 GHz PowerPC G5



2 GHz AMD Opteron



500 MHz Ultrasparc IIe

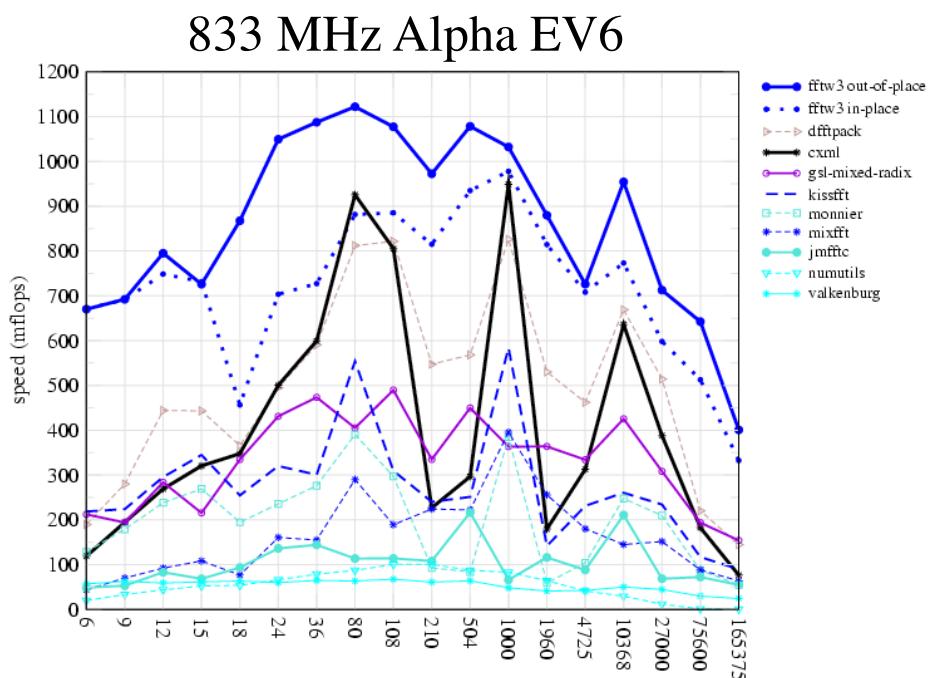
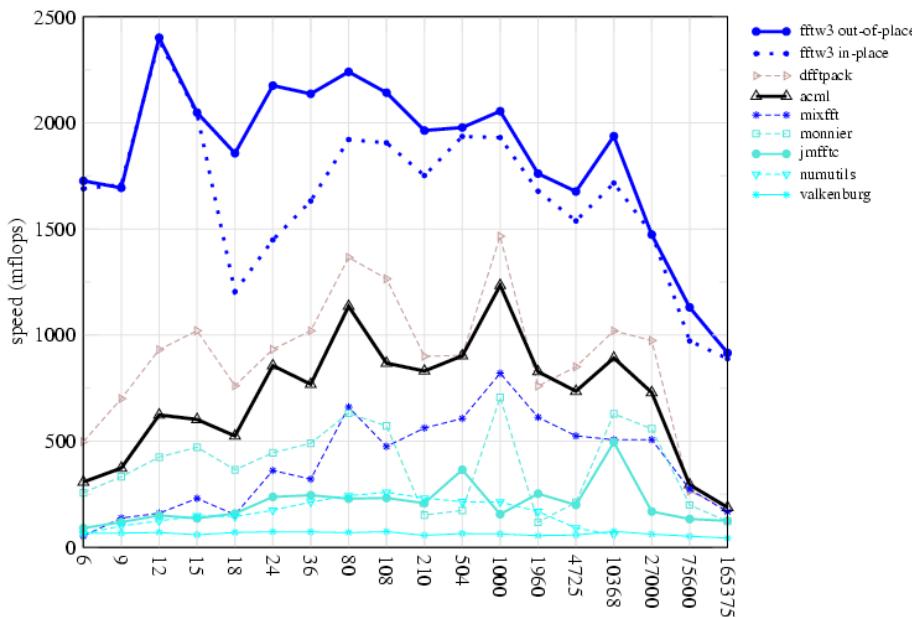


# FFTW performance

non-power-of-two sizes, double precision

unusual: non-power-of-two sizes  
receive as much optimization  
as powers of two

2 GHz AMD Opteron

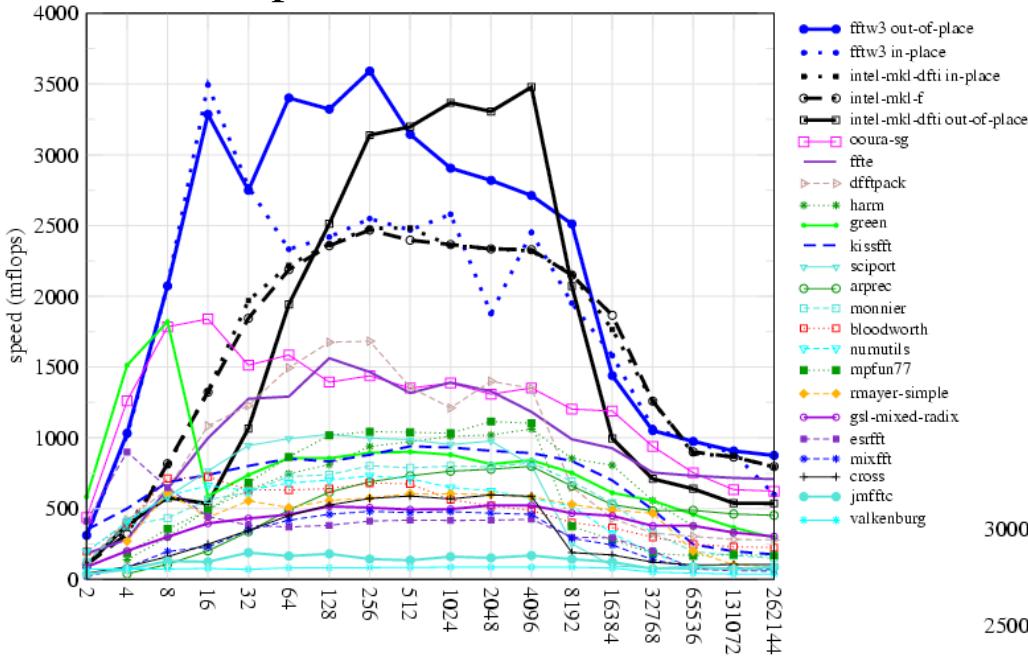


...because we  
let the code do the optimizing

# FFTW performance

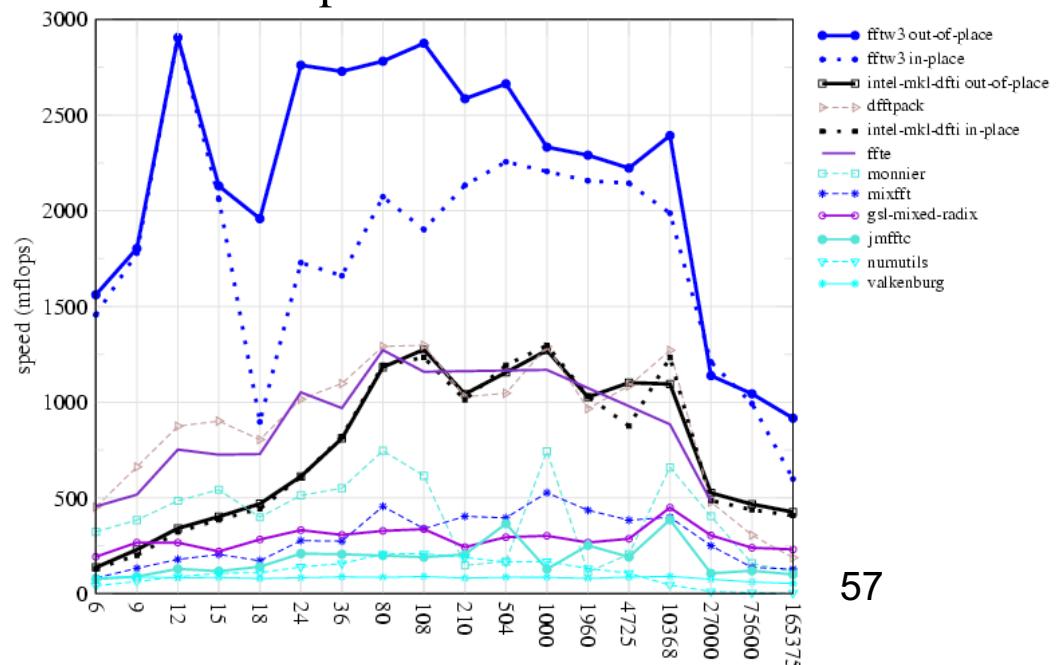
double precision, 2.8GHz Pentium IV: 2-way SIMD (SSE2)

powers of two



exploiting CPU-specific  
SIMD instructions  
(rewriting the code)  
is easy

non-powers-of-two



...because we  
let the code write itself

03/22/2018

57

# Why is FFTW fast?

three unusual features

---

FFTW implements many FFT algorithms:

A planner picks the best composition  
by measuring the speed of different combinations.

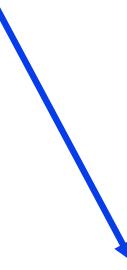
The resulting *plan* is executed  
with explicit recursion:  
enhances *locality*

The base cases of the recursion are codelets:  
highly-optimized dense code  
automatically generated by a special-purpose “compiler”

## FFTW is easy to use

---

```
{  
    complex x[n];  
    plan p;  
  
    p = plan_dft_1d(n, x, x, FORWARD, MEASURE);  
    ...  
    execute(p); /* repeat as needed */  
    ...  
    destroy_plan(p);  
}
```



**Key fact:** usually,  
many transforms of same size  
are required.

# Why is FFTW fast?

three unusual features

---

3

FFTW implements many FFT algorithms:

A planner picks the best composition  
by measuring the speed of different combinations.

1

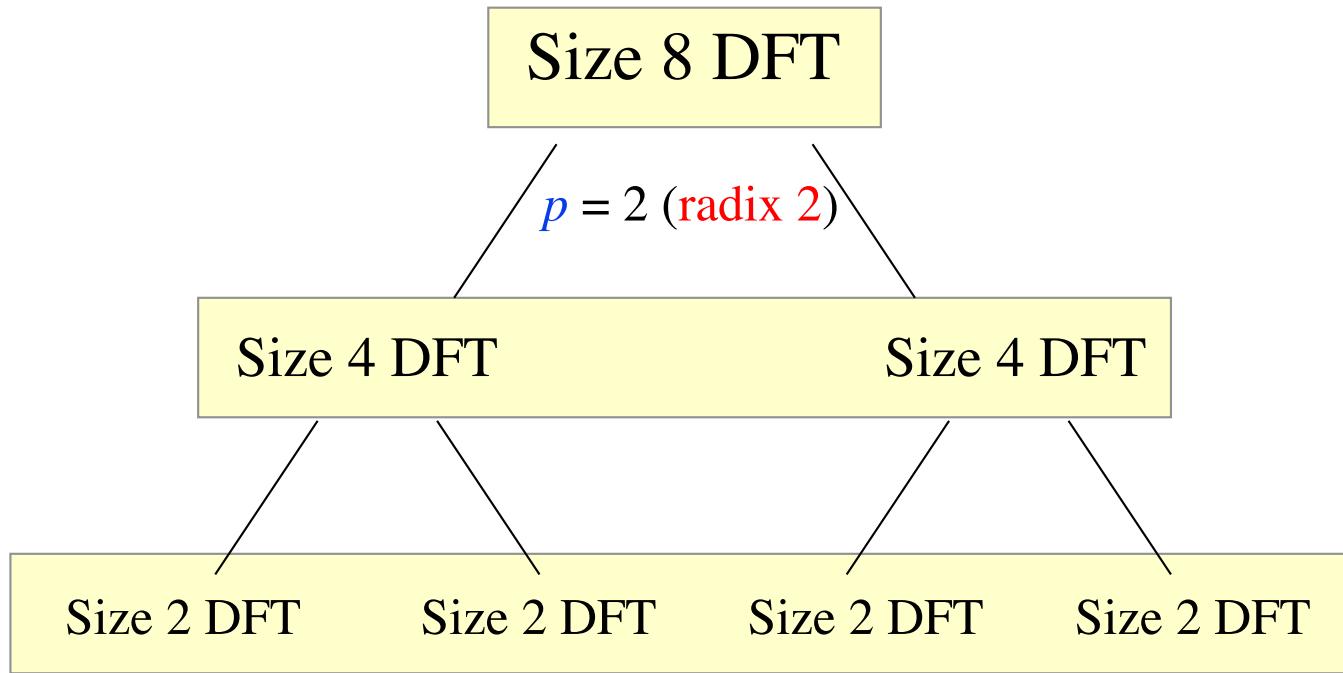
The resulting *plan* is executed  
with explicit **recursion**:  
*enhances locality*

2

The base cases of the recursion are **codelets**:  
highly-optimized dense code  
automatically generated by a special-purpose “compiler”

# FFTW Uses Natural Recursion

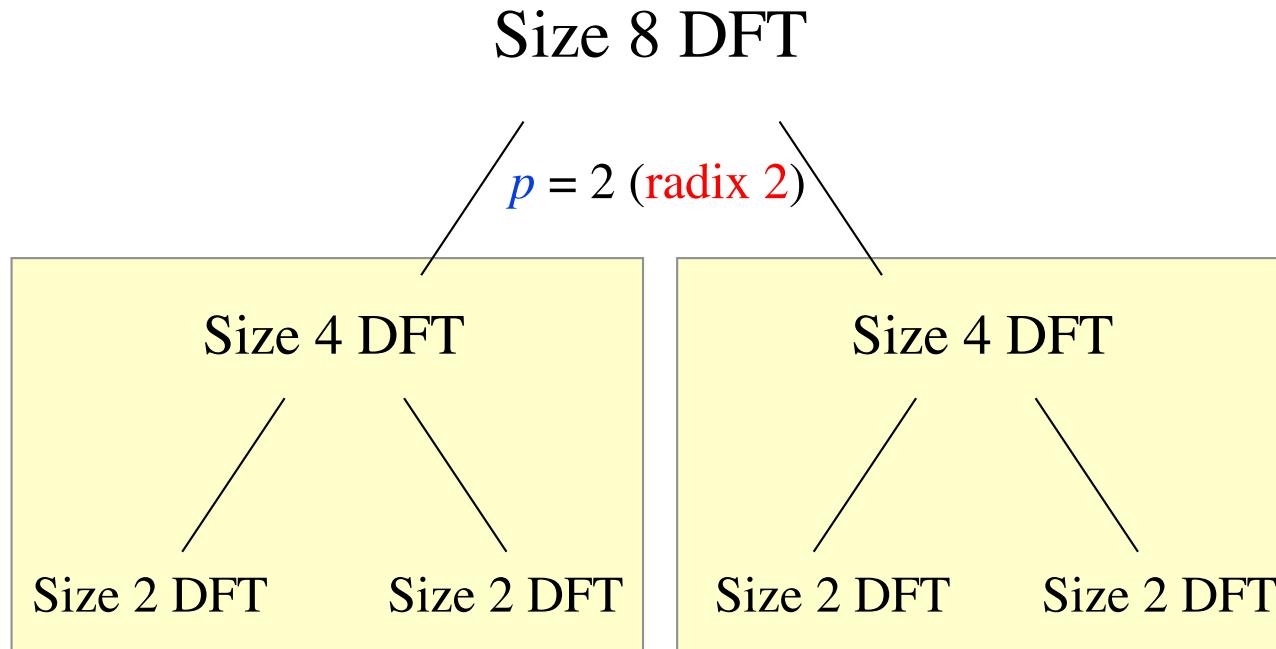
---



But **traditional** implementation is **non-recursive**,  
**breadth-first** traversal:  
 $\log_2 n$  passes over **whole** array

## Traditional cache solution: Blocking

---



breadth-first, but with *blocks* of size = cache

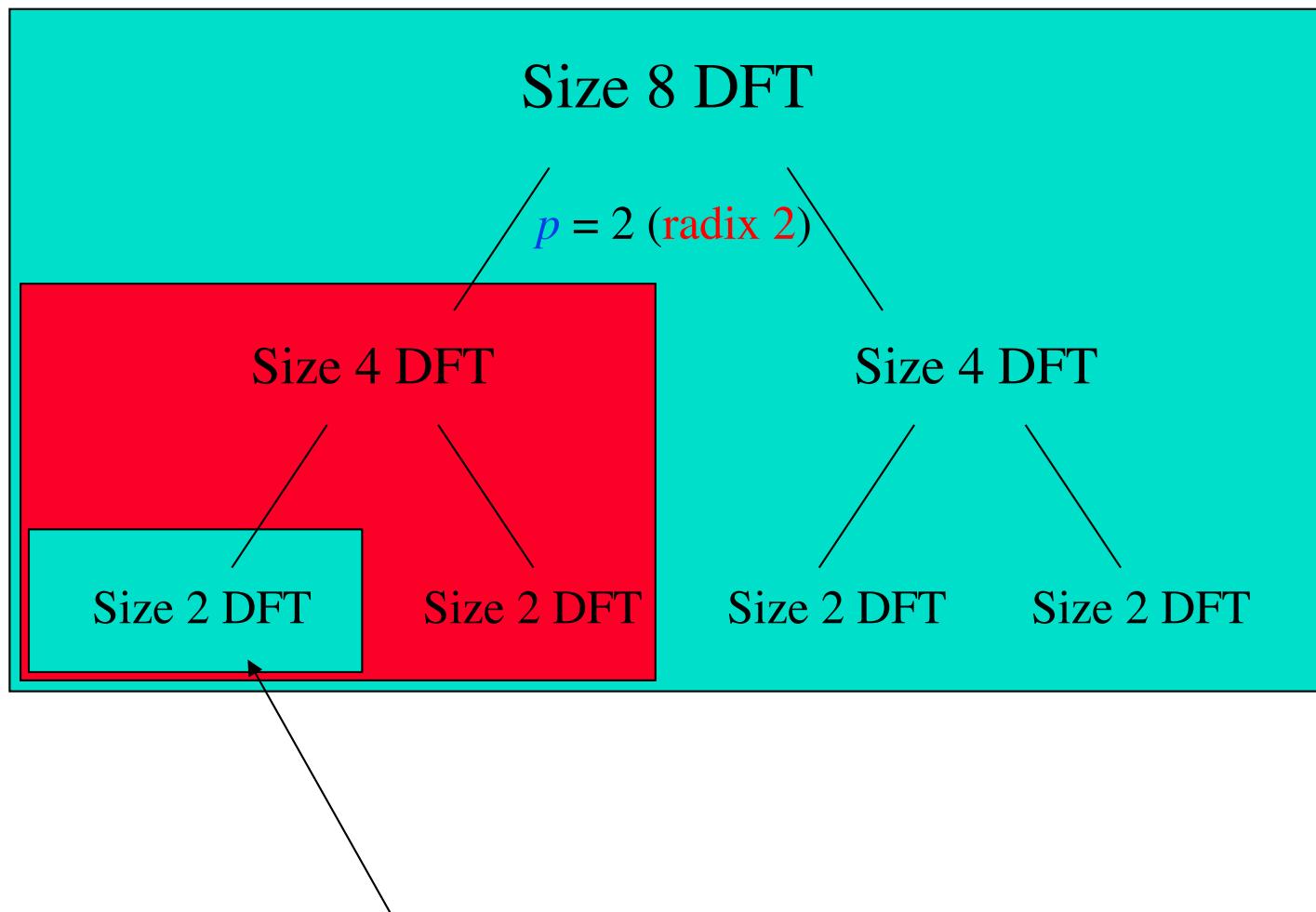
...requires program specialized for cache size

# Recursive Divide & Conquer is Good

---

(depth-first traversal)

[Singleton, 1967]



eventually small enough to fit in cache  
...no matter what size the cache is

# Cache Obliviousness

---

- A cache-oblivious algorithm does not know the cache size
  - it can be optimal for any machine & for all levels of cache simultaneously
- Exist for many other algorithms, too [Frigo *et al.* 1999]
  - all via the recursive divide & conquer approach

# Why is FFTW fast?

three unusual features

---

3

FFTW implements many FFT algorithms:

A planner picks the best composition  
by measuring the speed of different combinations.

1

The resulting *plan* is executed  
with explicit recursion:  
enhances *locality*

2

The base cases of the recursion are **codelets**:  
highly-optimized dense code  
automatically generated by a **special-purpose “compiler”**

## Spiral

---

- ° **Software/Hardware Generation for DSP Algorithms**
- ° **Autotuning not just for FFT, many other signal processing algorithms**
- ° **Autotuning not just for software implementation, hardware too**
- ° **More details at**
  - [www.spiral.net](http://www.spiral.net)
  - **On-line generators, papers available**

# Motifs – so far this semester

---

The Motifs (formerly “Dwarfs”) from  
“The Berkeley View” (Asanovic et al.)

**Motifs** form key computational patterns

