



# **DoD Sensor Processing: Applications and Supporting Software Technology**

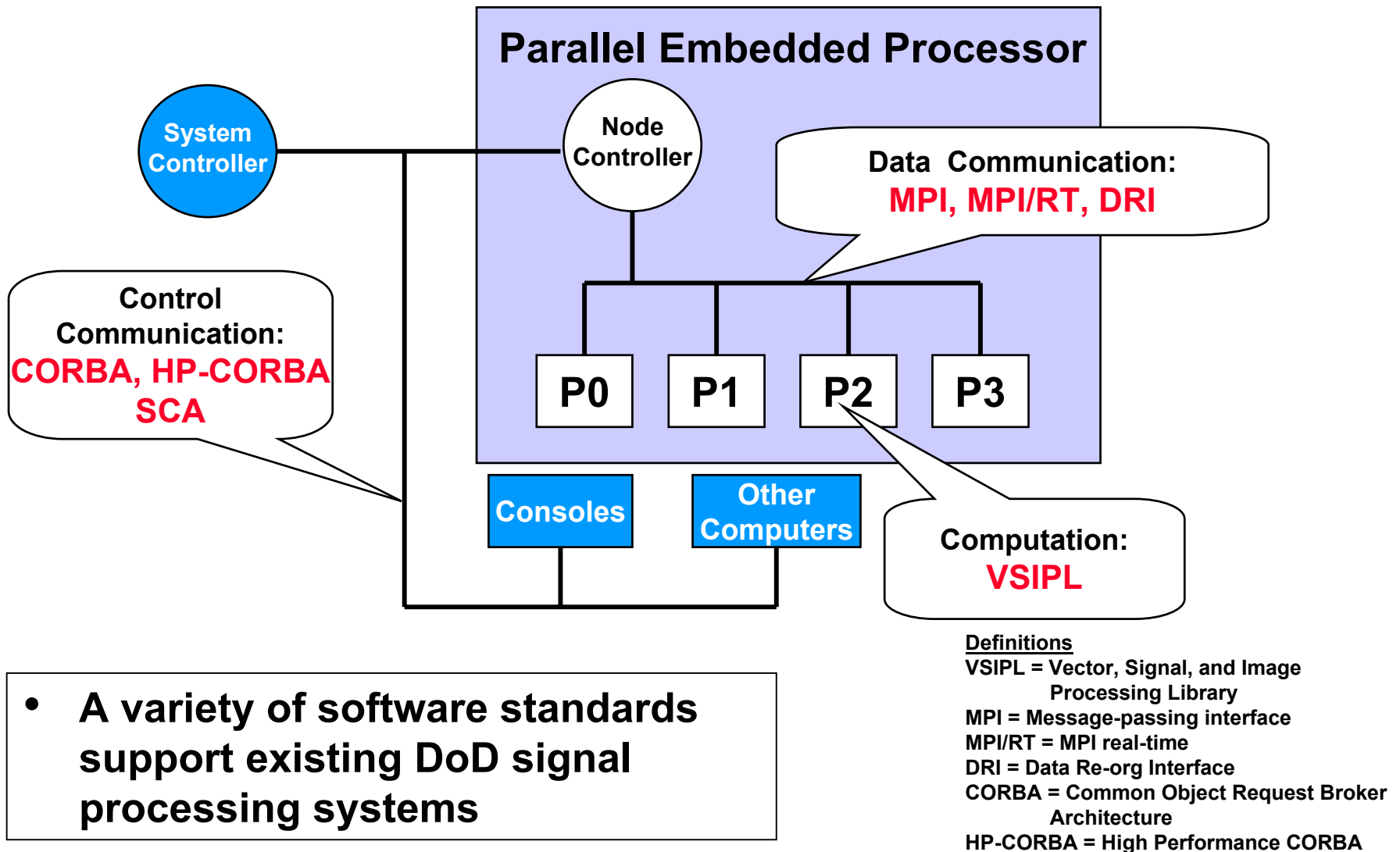
**Dr. Jeremy Kepner**

**MIT Lincoln Laboratory**

**This work is sponsored by the High Performance Computing Modernization Office under Air Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the author and are not necessarily endorsed by the United States Government.**

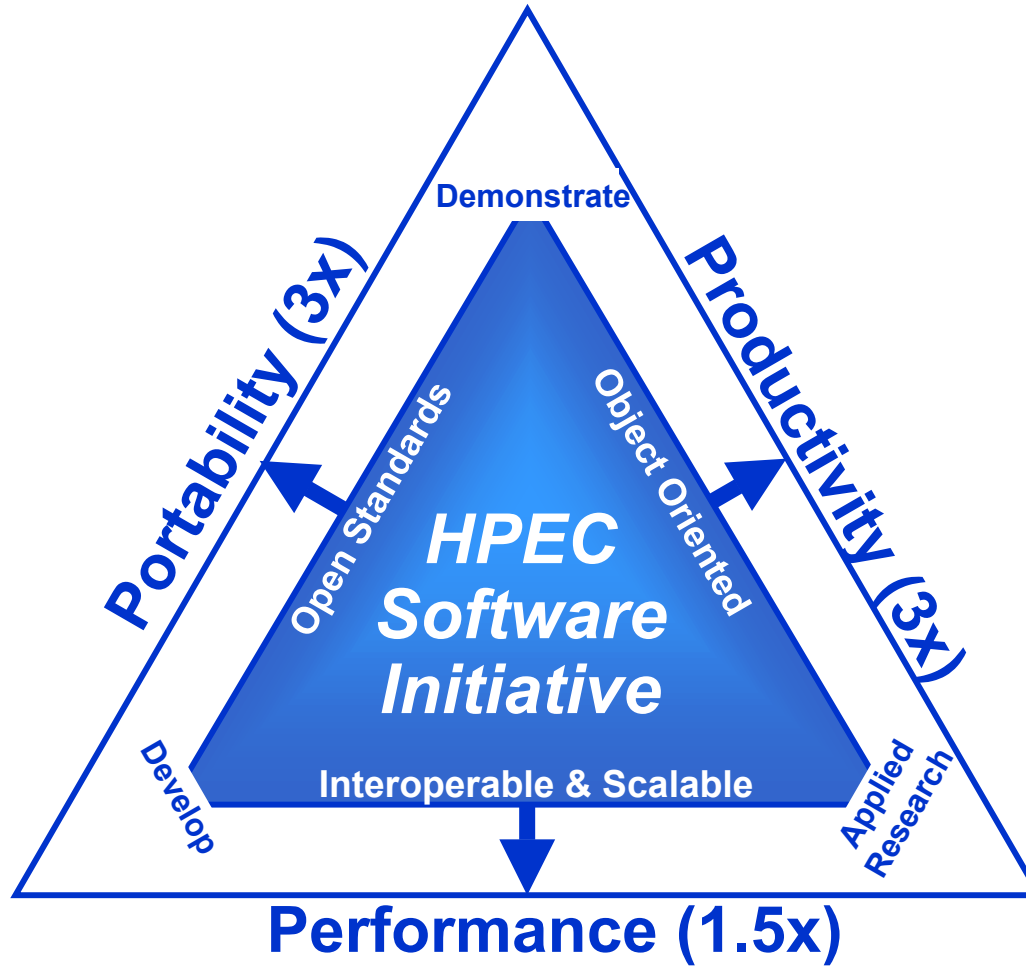
**MIT Lincoln Laboratory**

# Preamble: Existing Standards



# Preamble: Next Generation Standards

- Software Initiative Goal: transition research into commercial standards

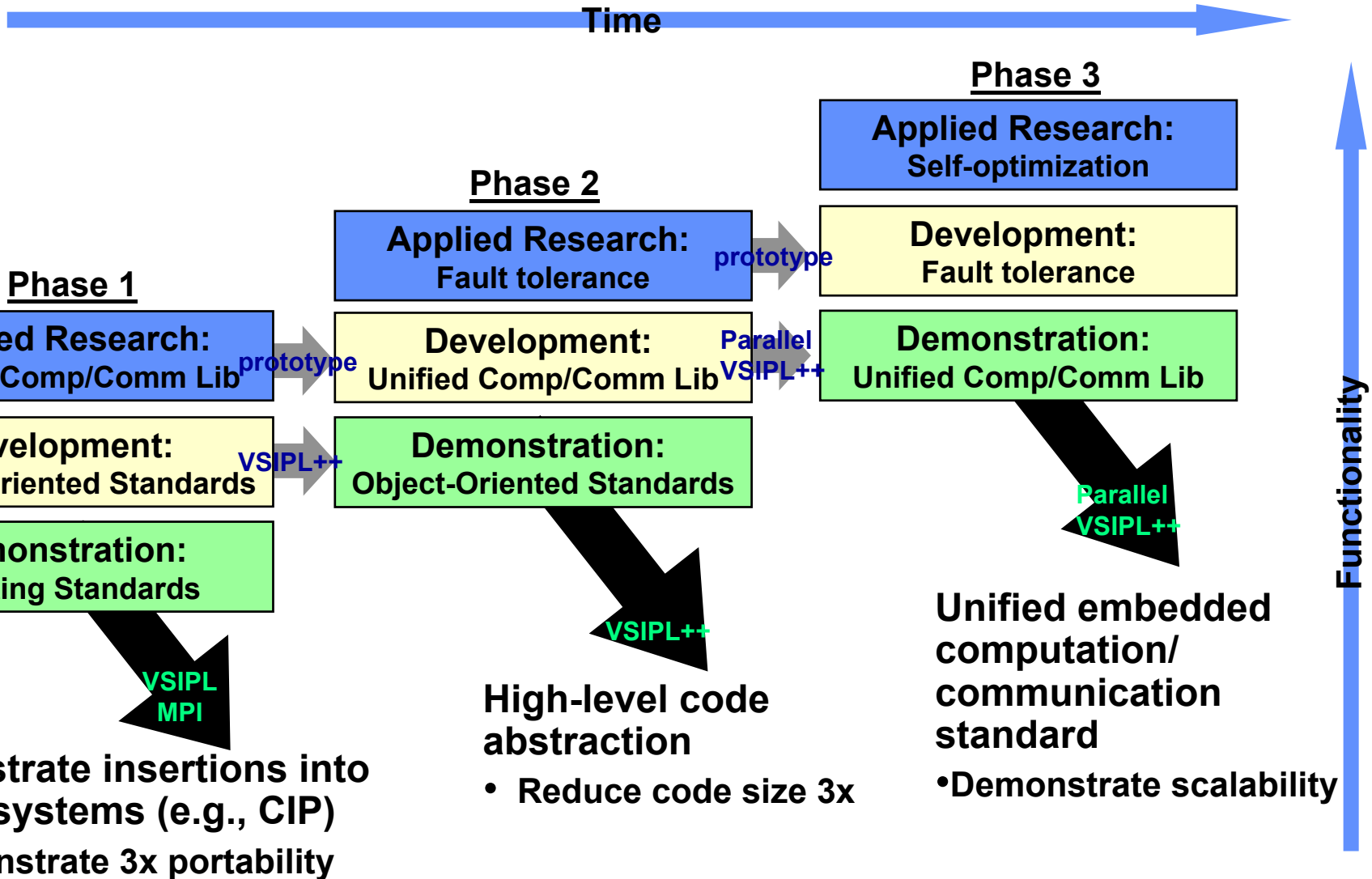


**Portability**  $\equiv$  lines-of-code changed to port/scale to new system

**Productivity**  $\equiv$  lines-of-code added to add new functionality

**Performance**  $\equiv$  computation and communication benchmarks

# HPEC-SI: VSIPL++ and Parallel VSIPL



# Preamble: The Links

High Performance Embedded Computing Workshop

<http://www.ll.mit.edu/HPEC>

High Performance Embedded Computing Software Initiative

<http://www.hpec-si.org/>

Vector, Signal, and Image Processing Library

<http://www.vsipl.org/>

MPI Software Technologies, Inc.

<http://www.mpi-softtech.com/>

Data Reorganization Initiative

<http://www.data-re.org/>

CodeSourcery, LLC

<http://www.codesourcery.com/>

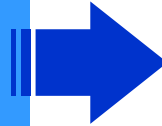
MatlabMPI

<http://www.ll.mit.edu/MatlabMPI>

# Outline

---

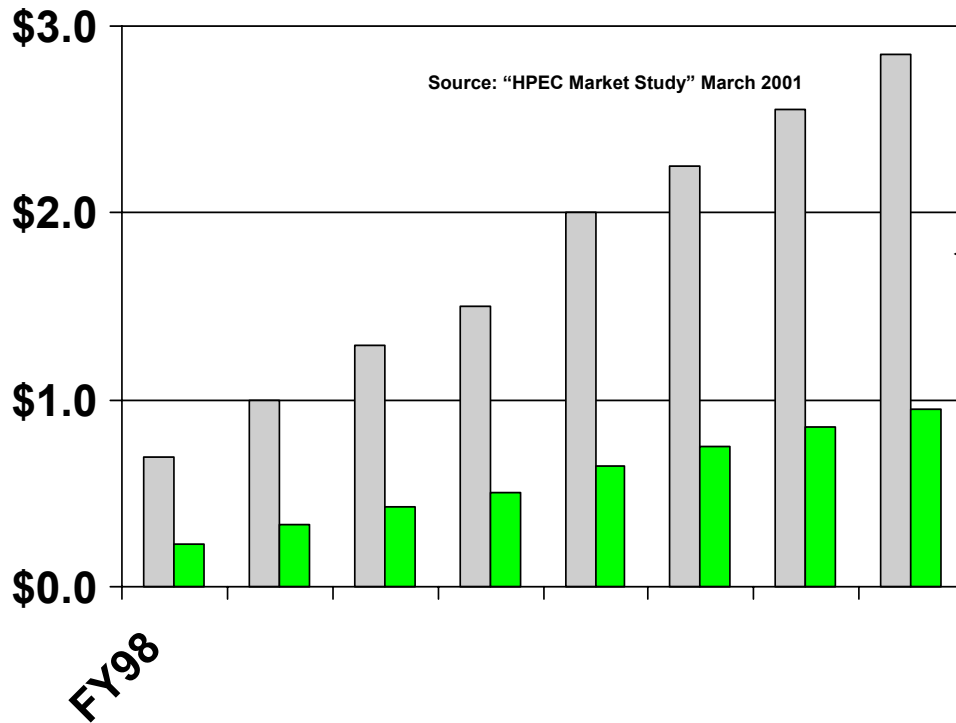
- **Introduction**



- *DoD Needs*
- *Parallel Stream Computing*
- *Basic Pipeline Processing*

- Processing Algorithms
- Parallel System Analysis
- Software Frameworks
- Summary

# Why Is DoD Concerned with Embedded Software?



**Estimated DoD expenditures for embedded signal and image processing hardware and software (\$B)**

- COTS acquisition practices have shifted the burden from “point design” hardware to “point design” software (i.e. COTS HW requires COTS SW)
- Software costs for embedded systems could be reduced by one-third with improved programming models, methodologies, and standards

# Embedded Stream Processing

Wireless

Video

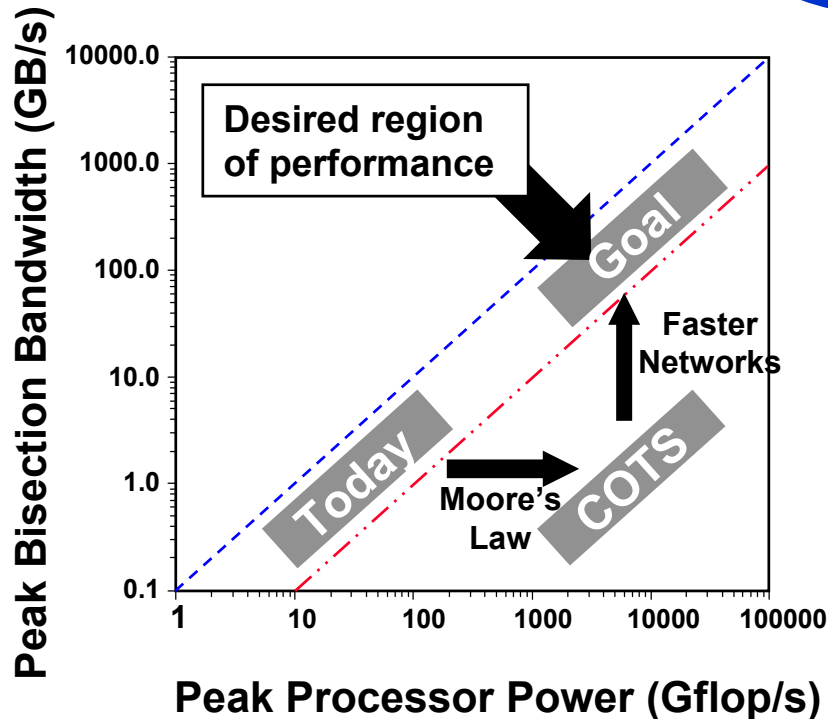
Medical

Radar

Sonar

Scientific

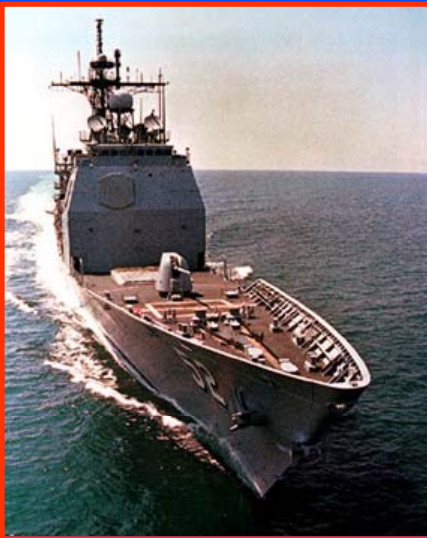
Encoding



Requires high performance computing *and* networking

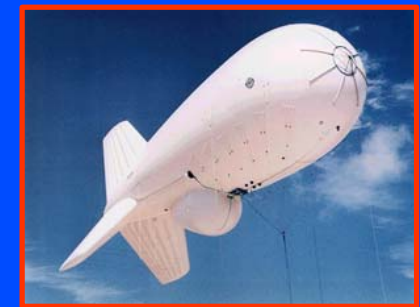
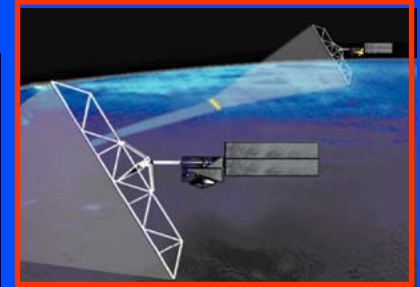


# Military Embedded Processing



REQUIREMENTS INCREASING BY AN ORDER OF MAGNITUDE EVERY 5 YEARS

**EMBEDDED PROCESSING REQUIREMENTS WILL EXCEED 10 TFLOPS IN THE 2005-2010 TIME FRAME**



- Signal processing drives computing requirements
- Rapid technology insertion is critical for sensor dominance



# Military Query Processing

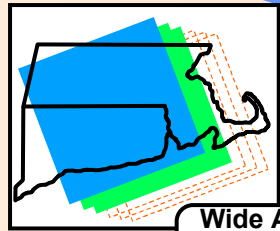
Sensors

High Speed Networks

Parallel Computing

Software

Missions



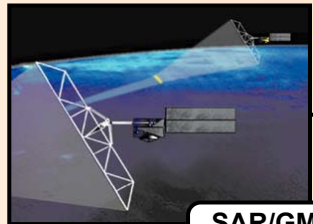
Wide Area Imaging



HPCMP



Targeting



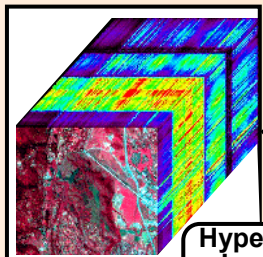
SAR/GMTI



ASCI



Force Location



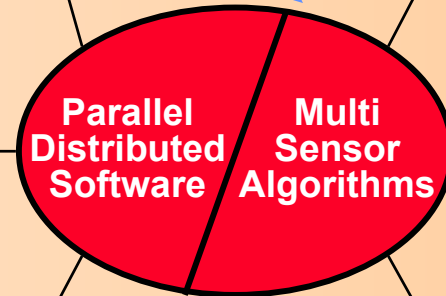
Hyper Spec Imaging



HPCC



Infrastructure Assessment

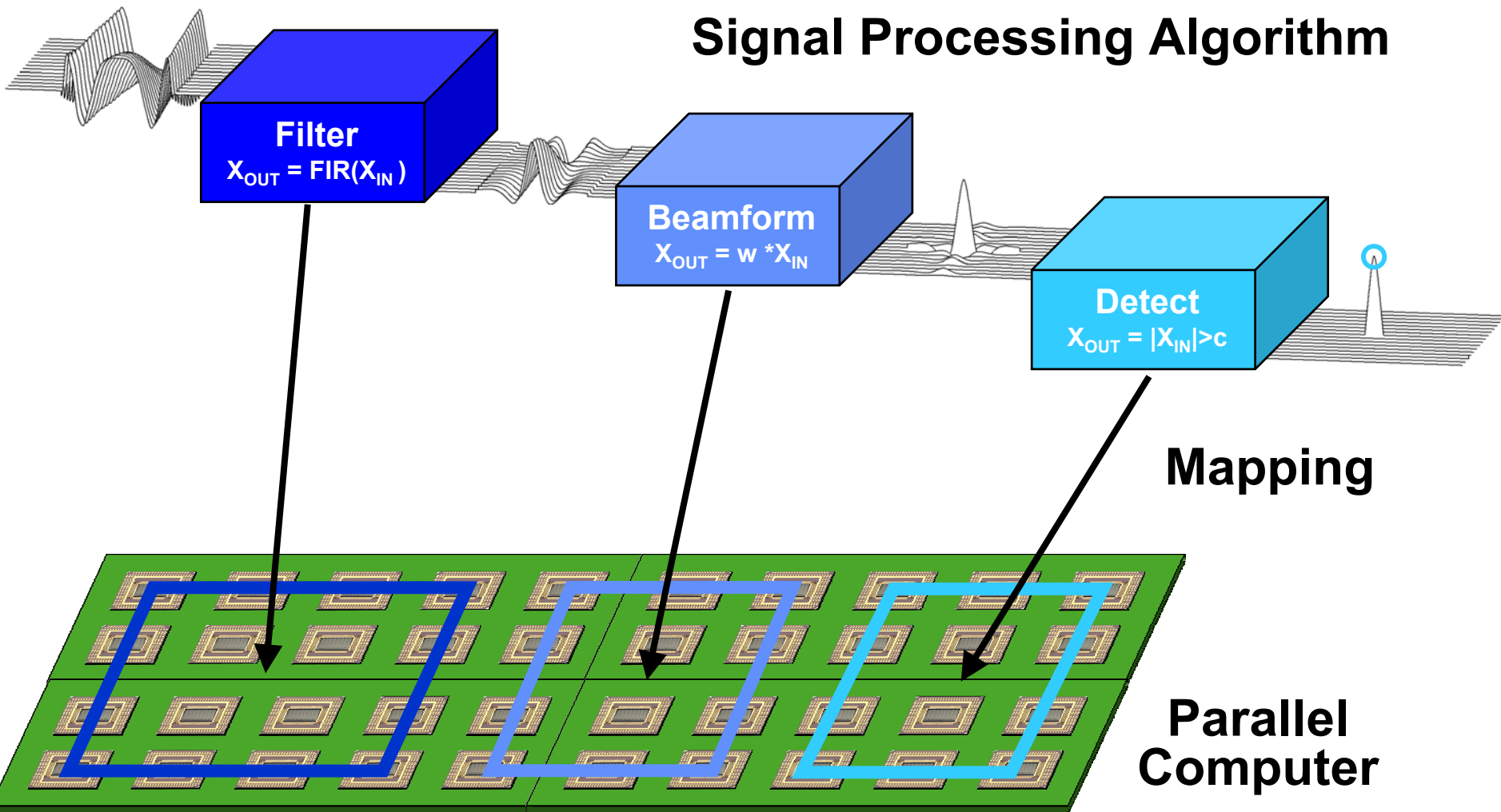


BoSSNET

- Highly distributed computing
- Fewer very large data movements

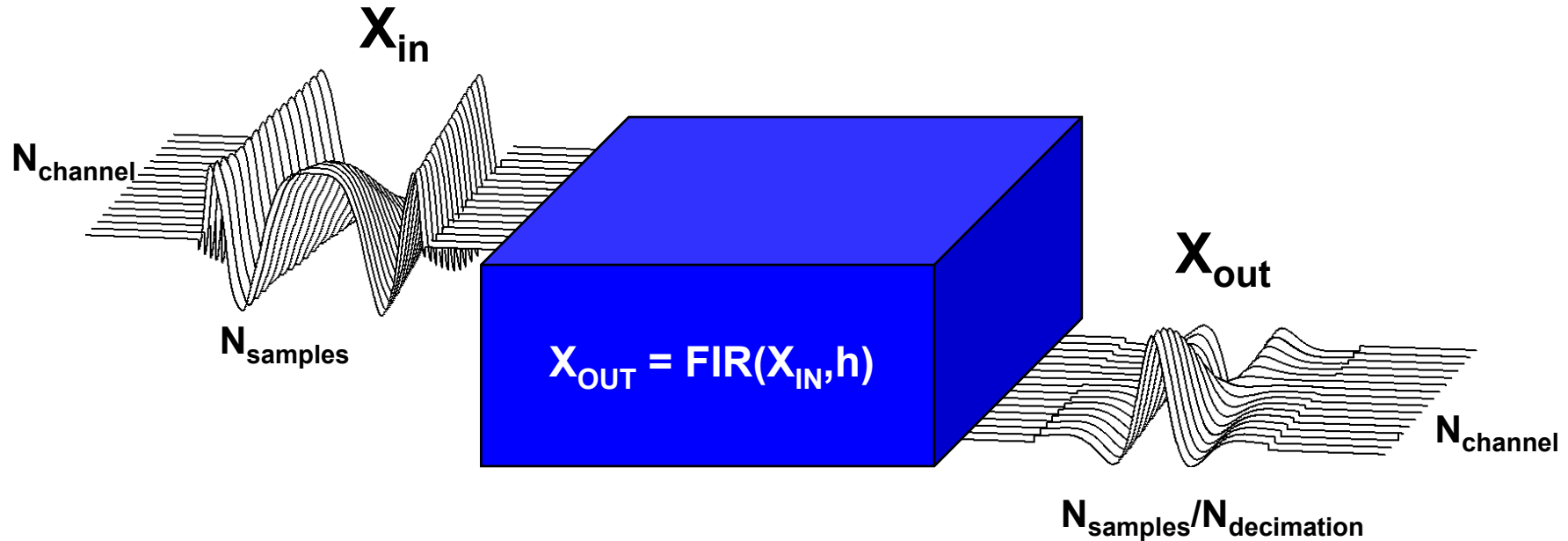
# Parallel Pipeline

## Signal Processing Algorithm



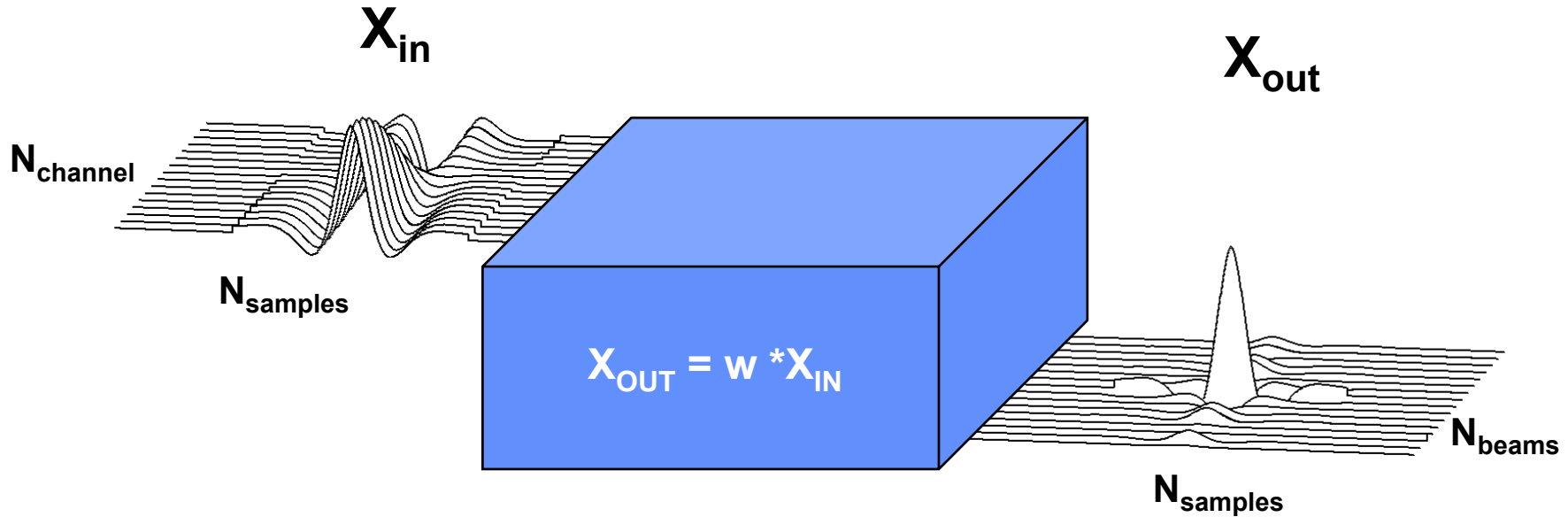
- Data Parallel within stages
- Task/Pipeline Parallel across stages

# Filtering



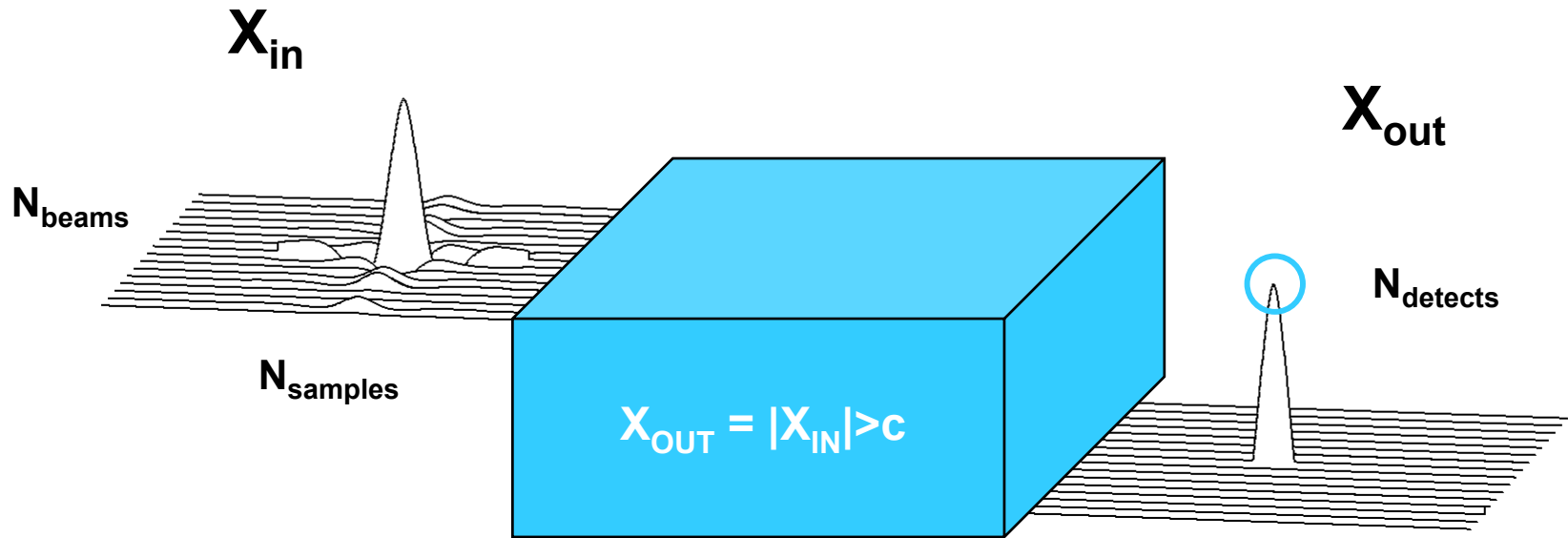
- Fundamental signal processing operation
- Converts data from wideband to narrowband via filter  
 $O(N_{samples} N_{channel} N_h / N_{decimation})$
- Degrees of parallelism:  $N_{channel}$

# Beamforming



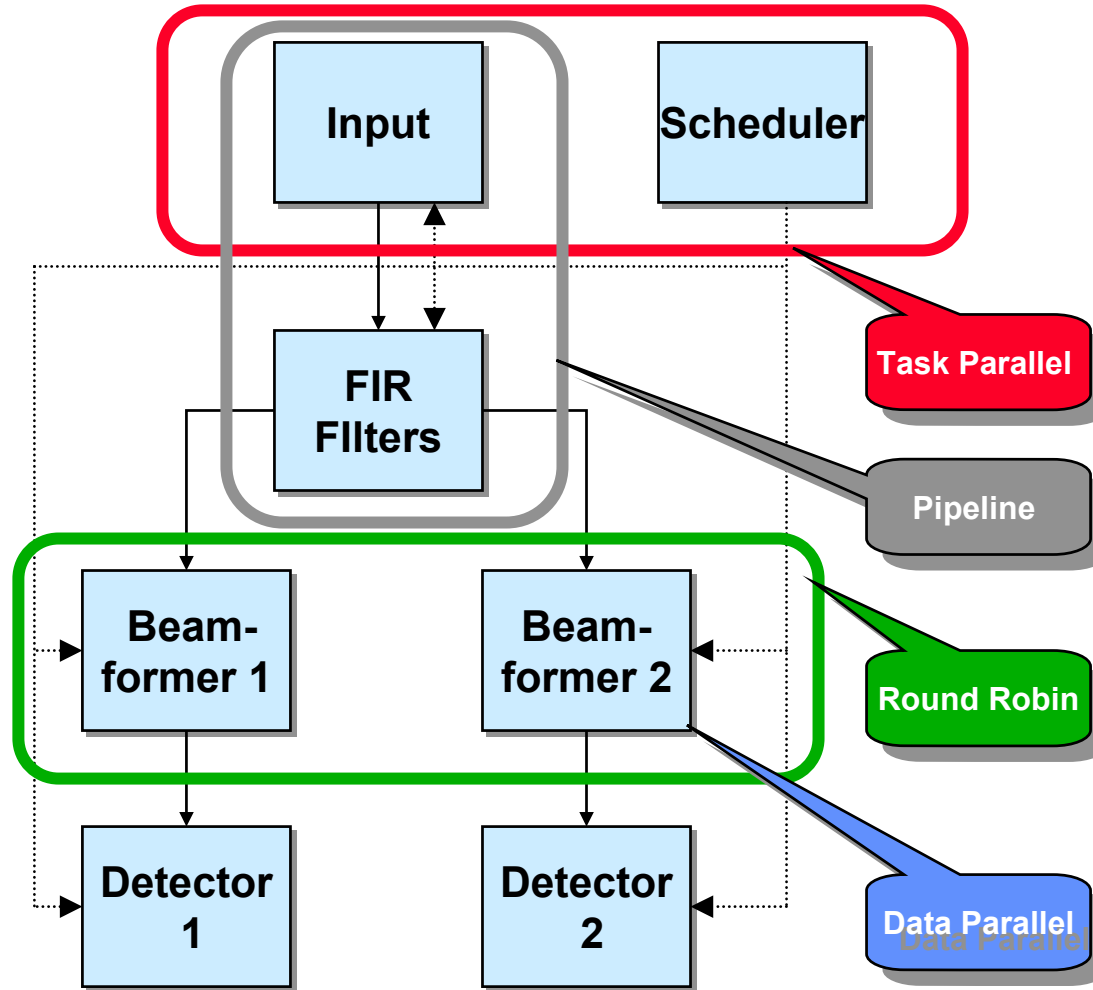
- Fundamental operation for all multi-channel receiver systems
- Converts data from channels to beams via matrix multiply  $O(N_{samples} N_{channel} N_{beams})$
- Key: weight matrix can be computed in advance
- Degrees of Parallelism:  $N_{samples}$

# Detection



- Fundamental operation for all processing chains
- Converts data from a stream to a list of detections via thresholding  $O(N_{samples} N_{beams})$
- Number detections is data dependent
- Degrees of parallelism:  $N_{beams}$   $N_{channels}$  or  $N_{detects}$

# Types of Parallelism

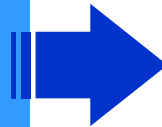


# Outline

---

- Introduction

- **Processing Algorithms**



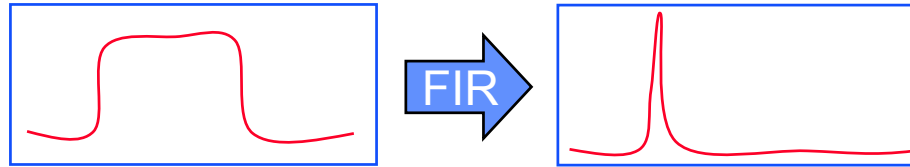
- *Filtering*
- *Beamforming*
- *Detection*

- Parallel System Analysis

- Software Frameworks

- Summary

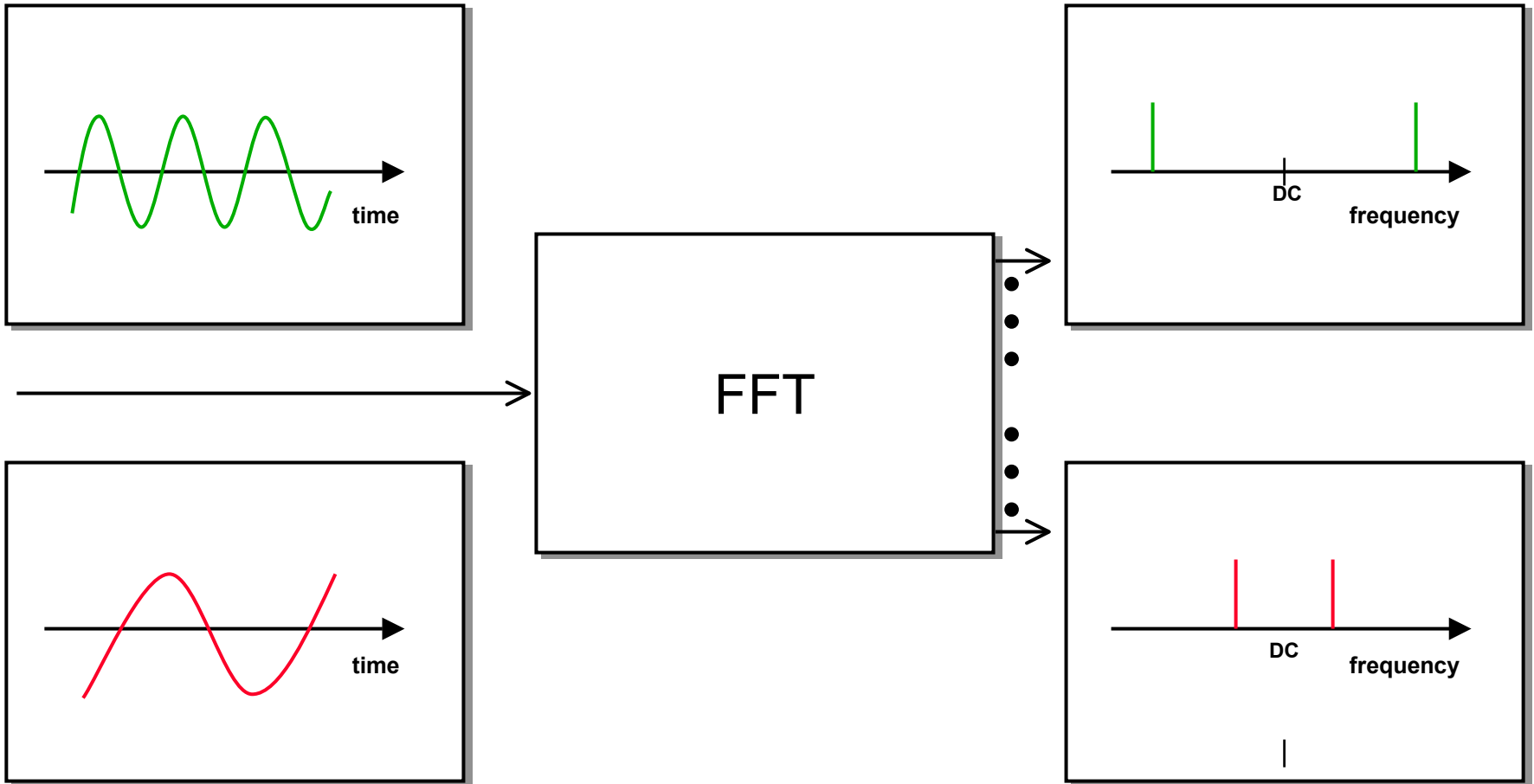
# FIR Overview



- **Uses:** pulse compression, equalization, ...
- **Formulation:**  $y = h \circ x$ 
  - $y$  = filtered data [#samples]
  - $x$  = unfiltered data [#samples]
  - $f$  = filter [#coefficients]
  - $\circ$  = convolution operator
- **Algorithm Parameters:** #channels, #samples, #coefficients, #decimation
- **Implementation Parameters:** Direct Sum or FFT based

# Basic Filtering via FFT

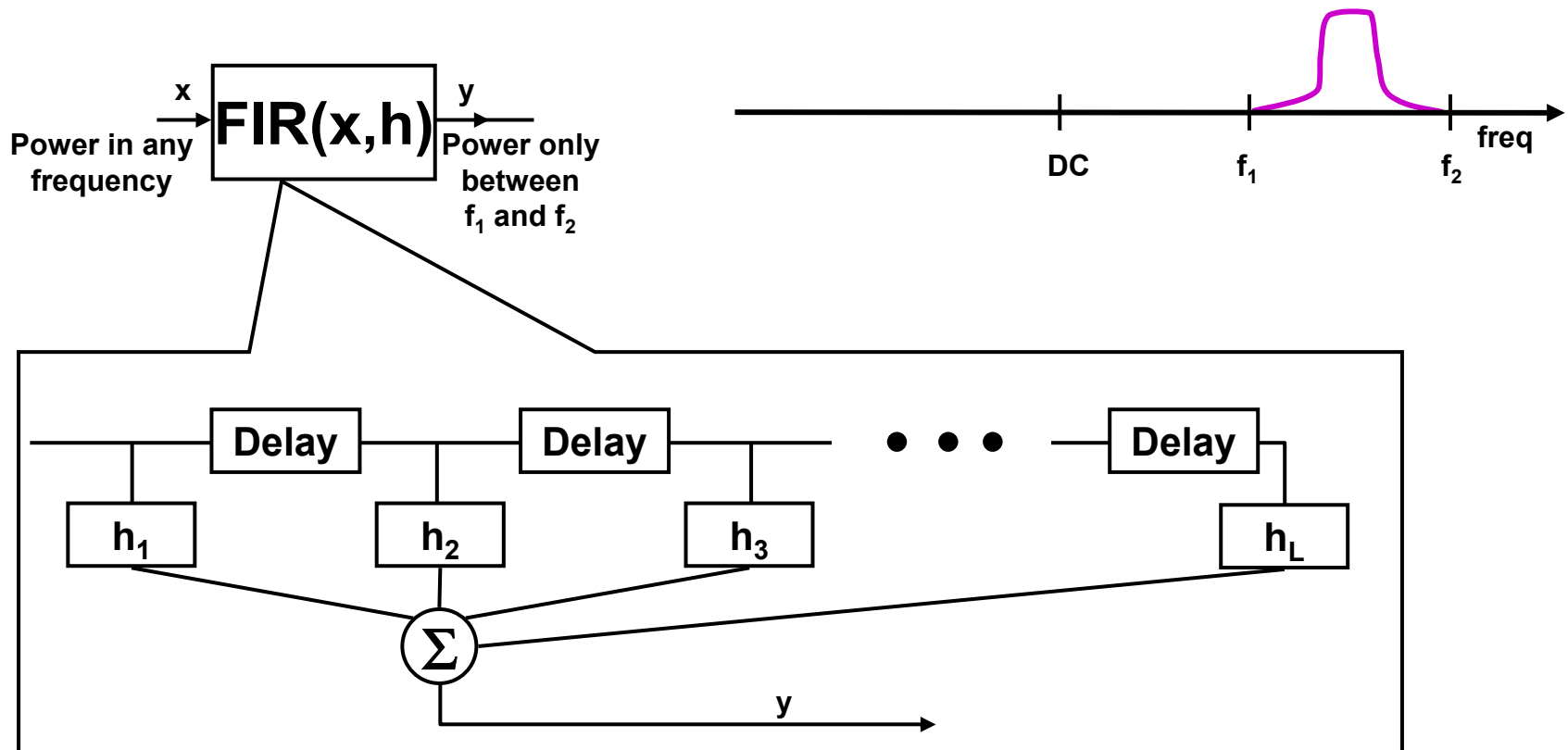
- **Fourier Transform (FFT) allows specific frequencies to be selected  $O(N \log N)$**



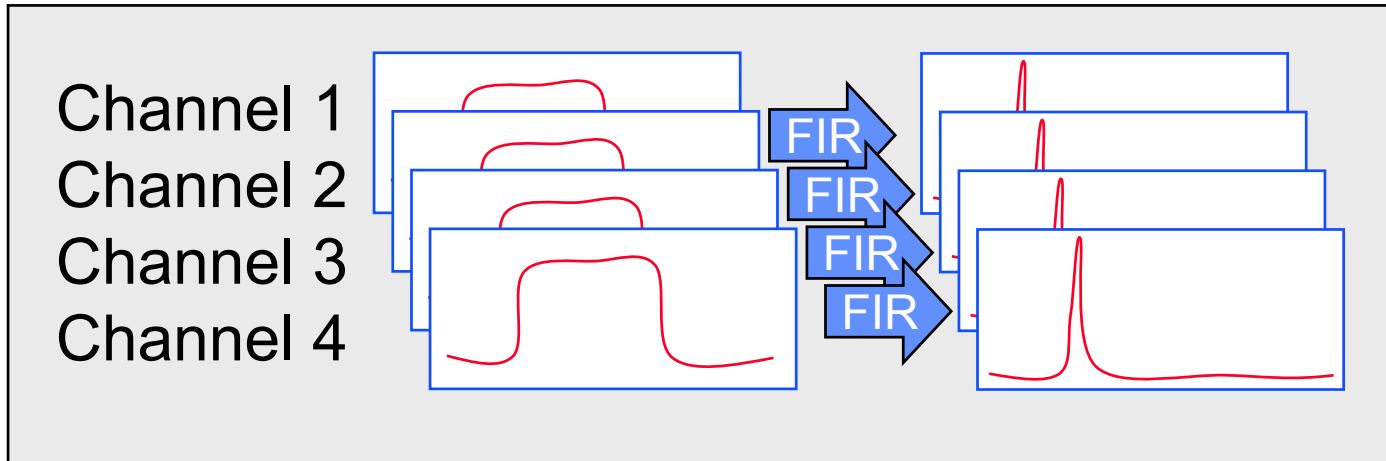
# Basic Filtering via FIR

- Finite Impulse Response (FIR) allows a range of frequencies to be selected  $O(N N_h)$

(Example: Band-Pass Filter)



# Multi-Channel Parallel FIR filter



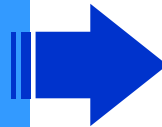
- **Parallel Mapping Constraints:**
  - **#channels MOD #processors = 0**
  - **1st parallelize across channels**
  - **2nd parallelize within a channel based on #samples and #coefficients**

# Outline

---

- Introduction

- **Processing Algorithms**



- *Filtering*
- *Beamforming*
- *Detection*

- Parallel System Analysis

- Software Frameworks

- Summary

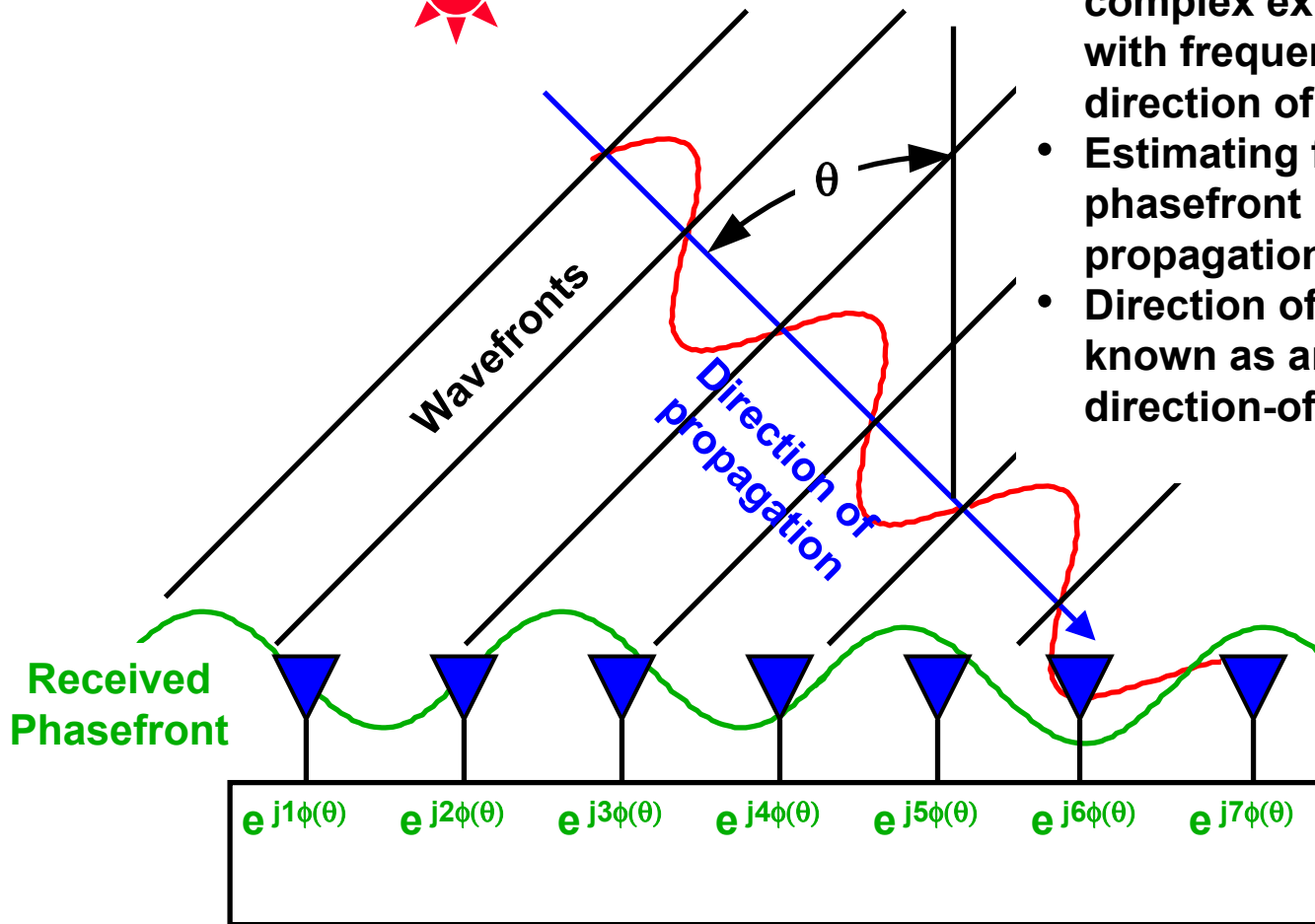
# Beamforming Overview



- **Uses: angle estimation**
- **Formulation:**  $y = w^H x$ 
  - $y$  = beamformed data [#samples x #beams]
  - $x$  = channel data [#samples x #channels]
  - $w$  = (tapered) steering vectors [#channels x #beams]
- **Algorithm Parameters: #channels, #samples, #beams, (tapered) steering vectors,**

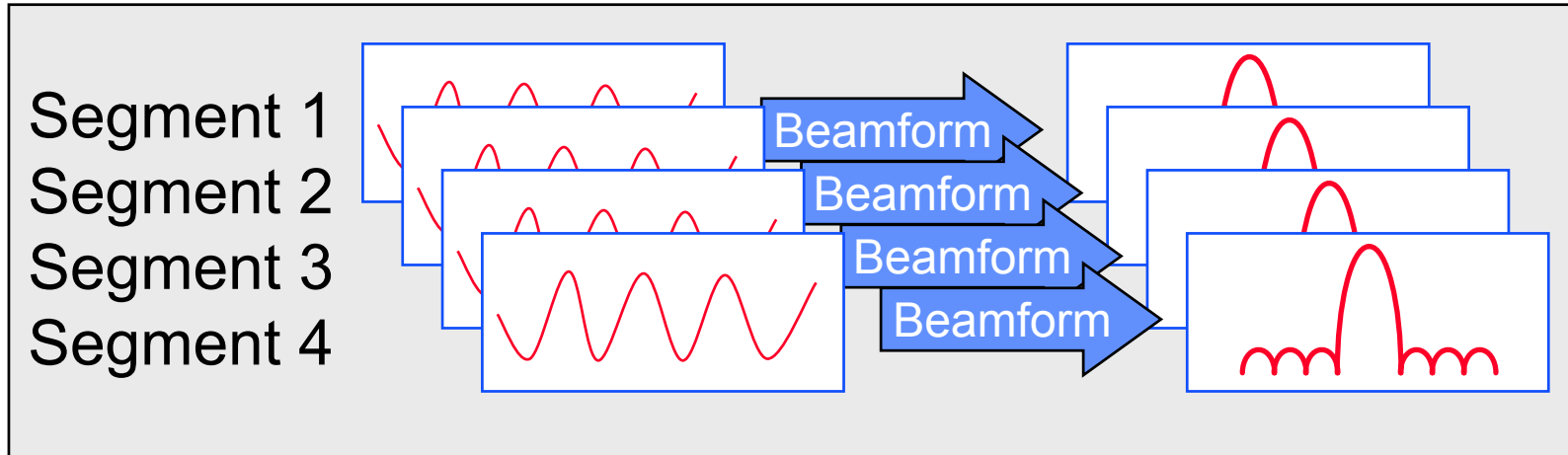
# Basic Beamforming Physics

Source



- Received phasefront creates complex exponential across array with frequency directly related to direction of propagation
- Estimating frequency of impinging phasefront indicates direction of propagation
- Direction of propagation is also known as angle-of-arrival (AOA) or direction-of arrival (DOA)

# Parallel Beamformer



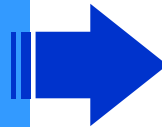
- **Parallel Mapping Constraints:**
  - **#segment MOD #processors = 0**
  - **1st parallelize across segments**
  - **2nd parallelize across beams**

# Outline

---

- Introduction

- **Processing Algorithms**



- *Filtering*
- *Beamforming*
- *Detection*

- Parallel System Analysis

- Software Frameworks

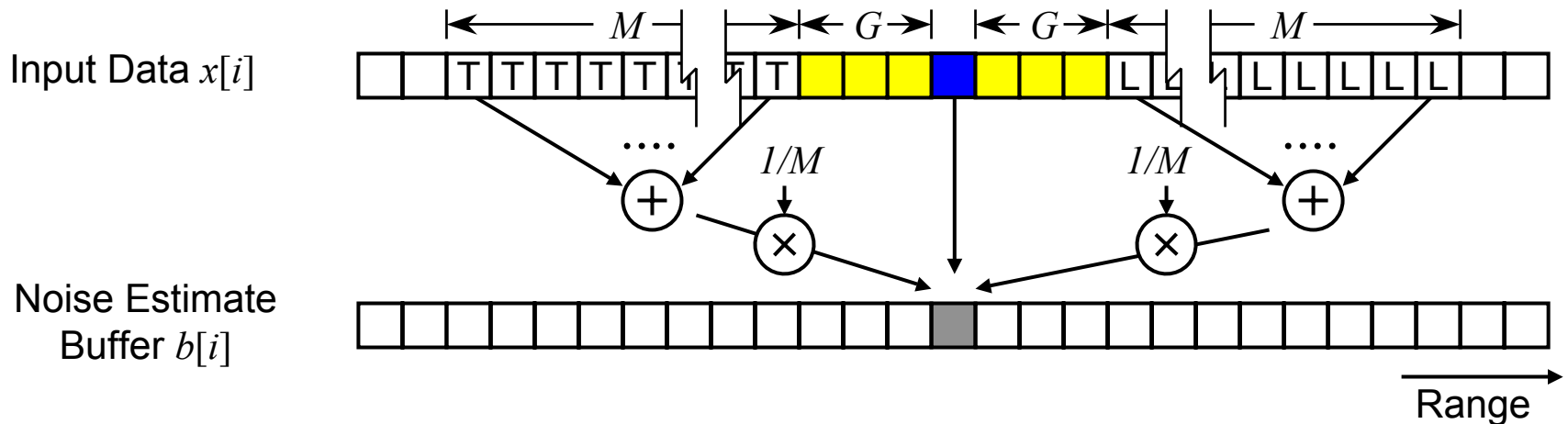
- Summary

# CFAR Detection Overview



- **Constant False Alarm Rate (CFAR)**
- **Formulation:**  $x[n] > T[n]$ 
  - $x[n]$  = cell under test
  - $T[n] = \text{Sum}(x_i)/2M$ ,  $N_{\text{gaurd}} < |i - N| < M + N_{\text{gaurd}}$
  - **Angle estimate: take ratio of beams; do lookup**
- **Algorithm Parameters: #samples, #beams, steering vectors, #noise samples, #max detects**
- **Implementation Parameters: Greatest Of, Censored Greatest Of, Ordered Statistics, ... Averaging vs Sorting**

# Two-Pass Greatest-Of Excision CFAR (First Pass)



- Range cell under test  $x[i]$
- Guard cells
- T Trailing training cells  $z_T[n]; n = 1, \dots, M$
- L Leading training cells  $z_L[n]; n = 1, \dots, M$

Excise

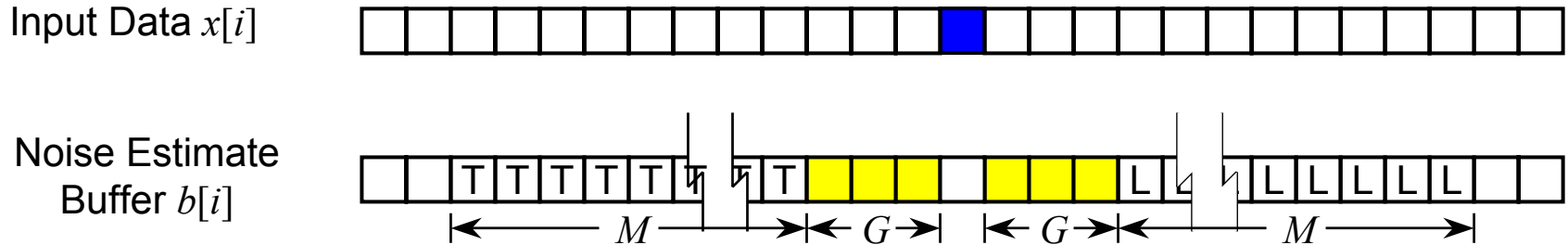
$$|x[i]|^2 \begin{matrix} > \\ < \end{matrix} \frac{T_1}{M} \times \max \left( \sum_{n=1}^M |z_T[n]|^2, \sum_{n=1}^M |z_L[n]|^2 \right)$$

Retain

Reference: S. L. Wilson, *Analysis of NRL's two-pass greatest-of excision CFAR*, Internal Memorandum, MIT Lincoln Laboratory, October 5 1998.

# Two-Pass Greatest-Of Excision CFAR


## (Second Pass)



 Cell under test  $x[i]$

 Trailing training cells  $z_T[n]; n = 1, \dots, M$

 Guard cells

 Leading training cells  $z_L[n]; n = 1, \dots, M$

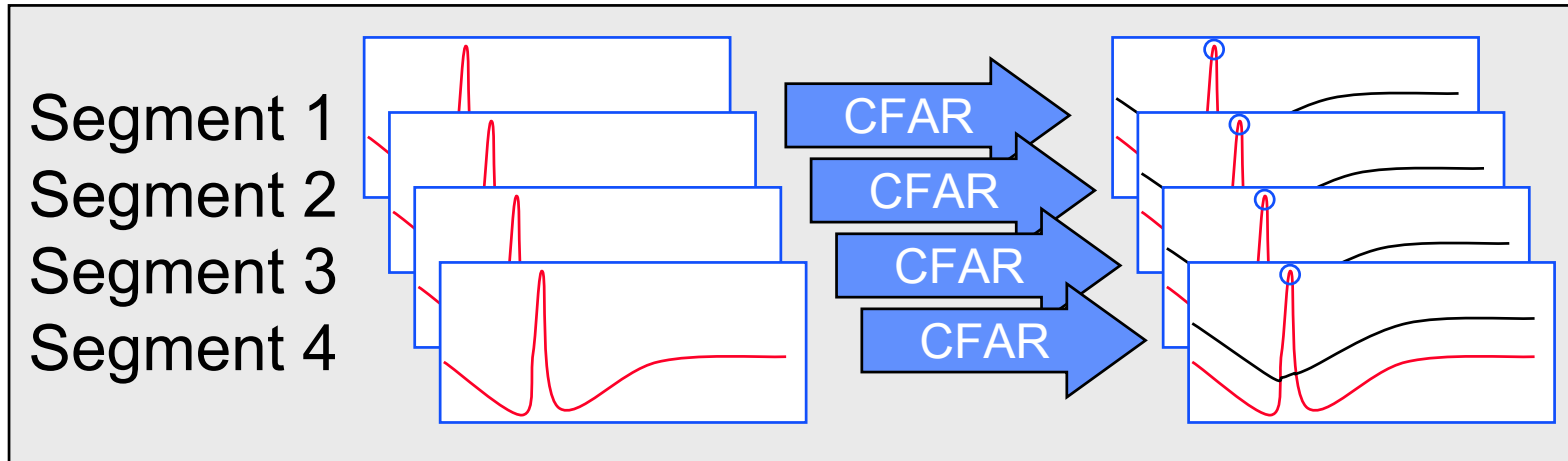
Target

$$|x[i]|^2 \begin{matrix} > \\ < \end{matrix} \frac{T_2}{M} \times \max \left( \sum_{n=1}^M |z_T[n]|^2, \sum_{n=1}^M |z_L[n]|^2 \right)$$

Noise

$$T_2 = f(M, T_1, P_{FA})$$

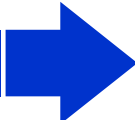
# Parallel CFAR Detection



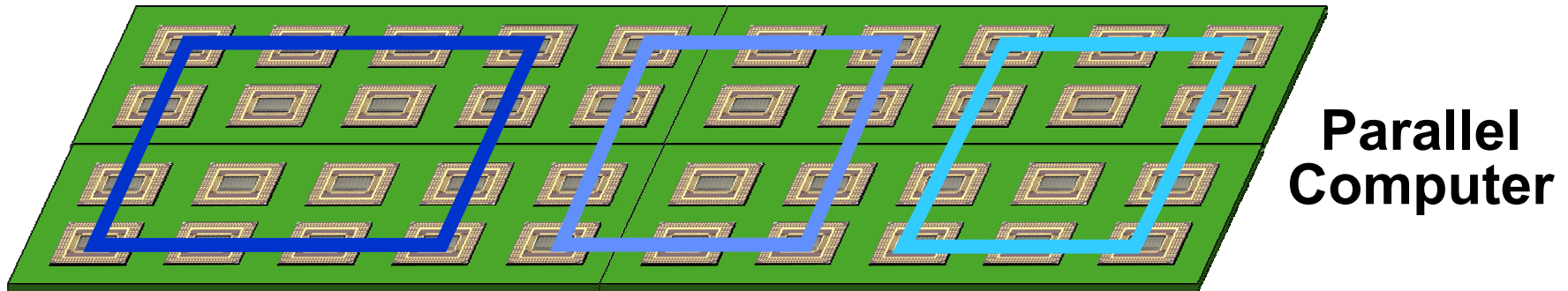
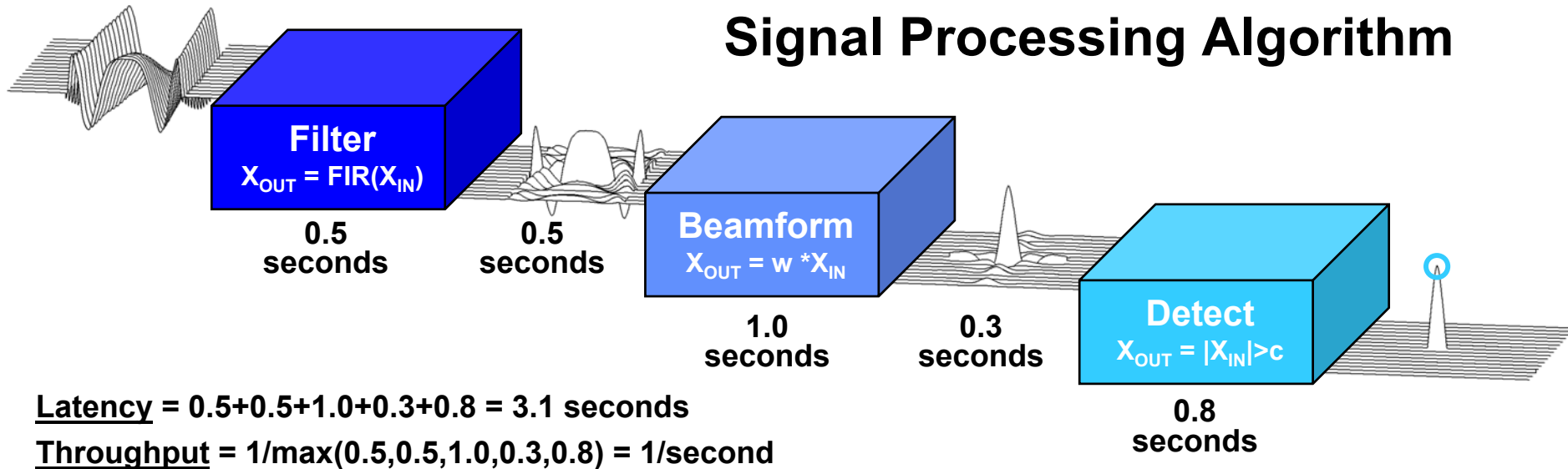
- **Parallel Mapping Constraints:**
  - **#segment MOD #processors = 0**
  - **1st parallelize across segments**
  - **2nd parallelize across beams**

# Outline

---

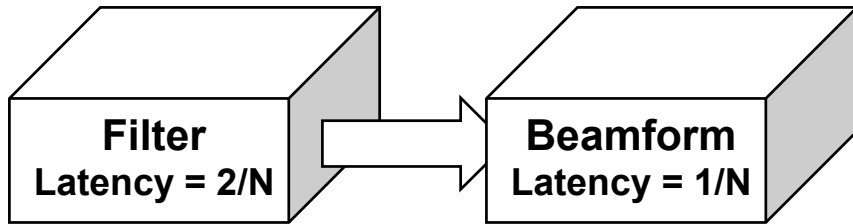
- Introduction
- Processing Algorithms
- **Parallel System Analysis** 
  - *Latency vs. Throughput*
  - *Corner Turn*
  - *Dynamic Load Balancing*
- Software Frameworks
- Summary

# Latency and throughput

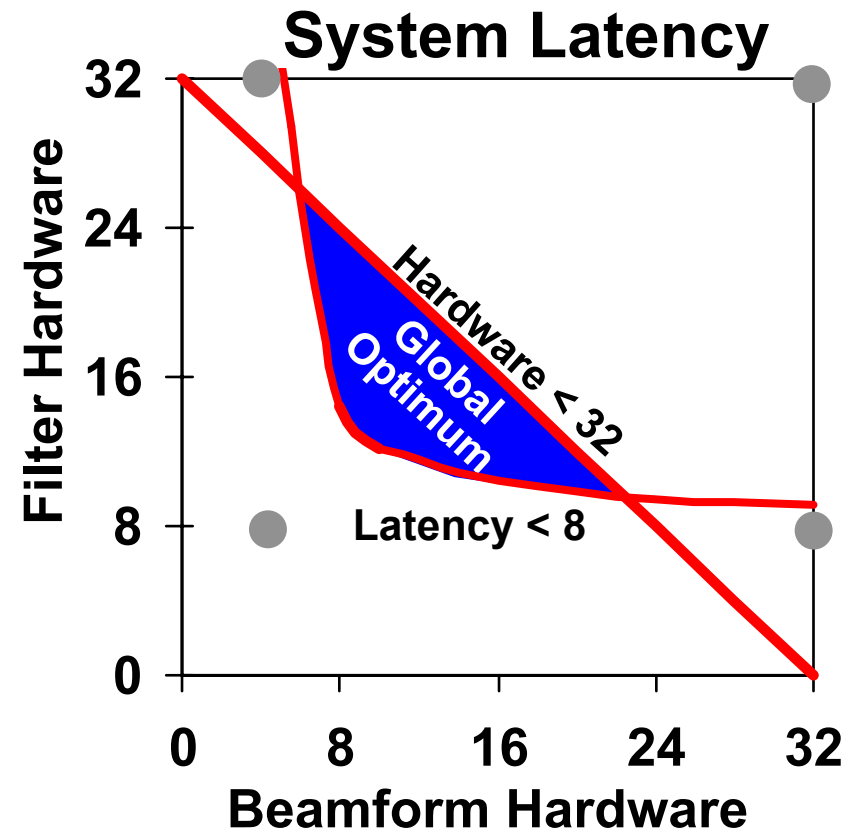
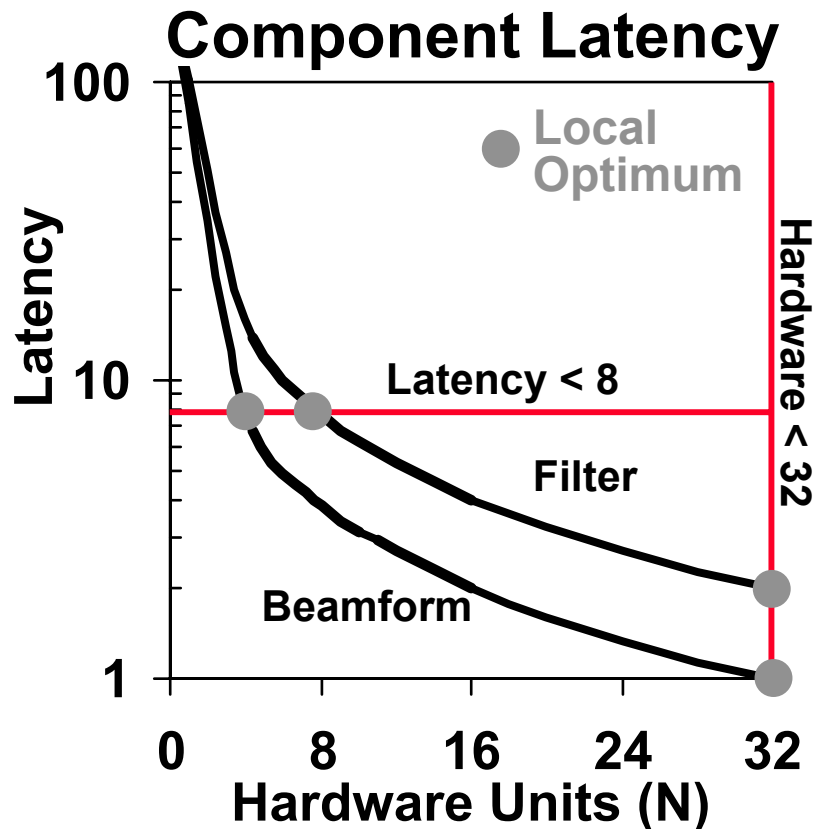


- **Latency:** total processing + communication time for one frame of data (sum of times)
- **Throughput:** rate at which frames can be input (max of times)

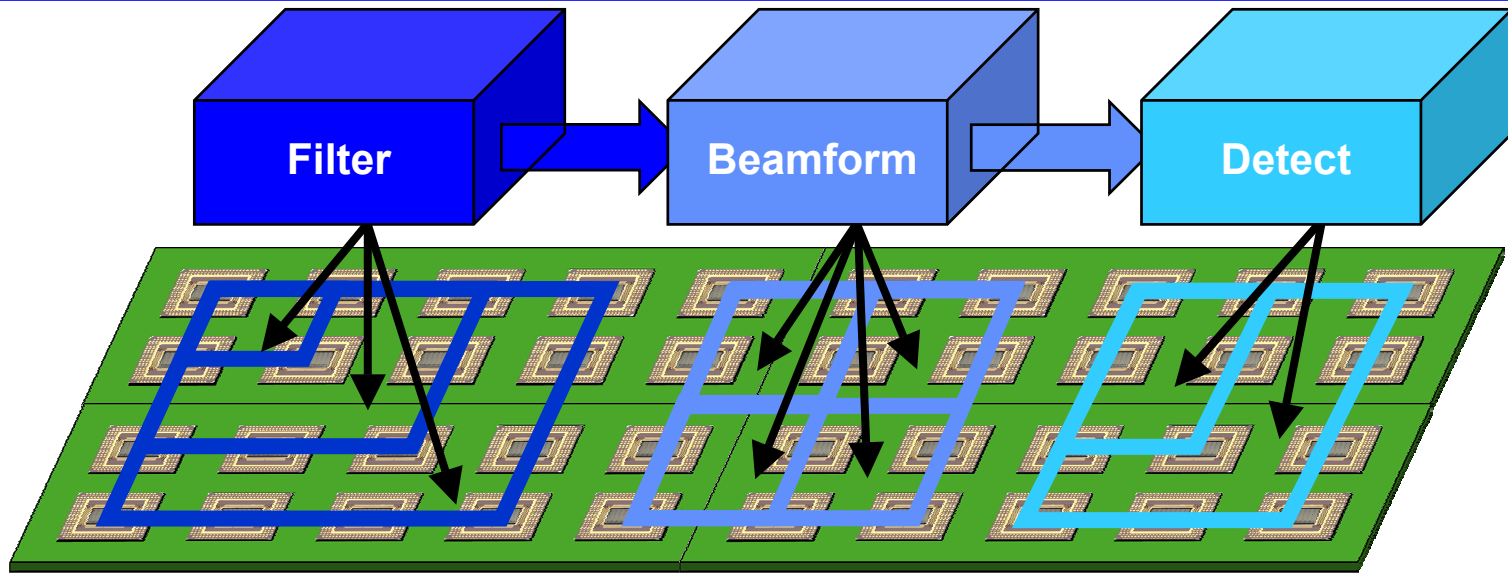
# Example: Optimum System Latency



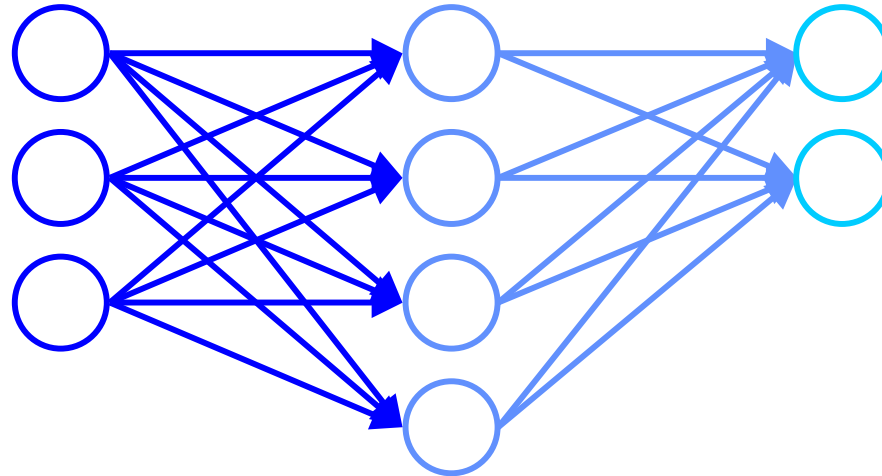
- Simple two component system
- Local optimum fails to satisfy **global constraints**
- Need system view to find **global optimum**



# System Graph



Node is a unique parallel mapping of a computation task

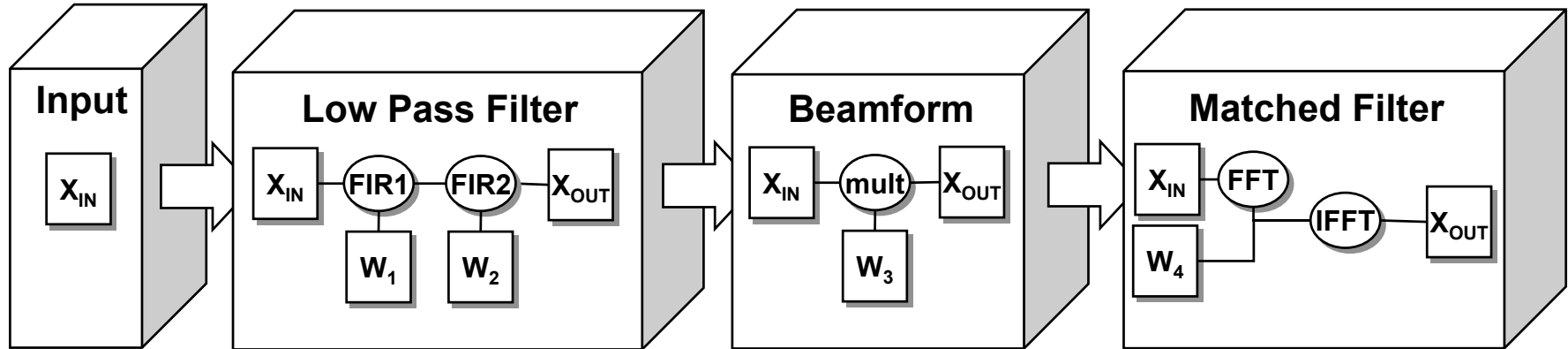


Edge is the conduit between a pair of parallel mappings

- System Graph can store the hardware resource usage of every possible Task & Conduit

# Optimal Mapping of Complex Algorithms

## Application



## Different Optimal Maps



Workstation



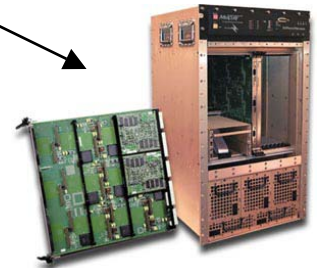
Intel Cluster



PowerPC Cluster



Embedded Board



Embedded Multi-computer

## Hardware

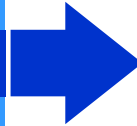
- Need to automate process of mapping algorithm to hardware

# Outline

---

- Introduction
- Processing Algorithms

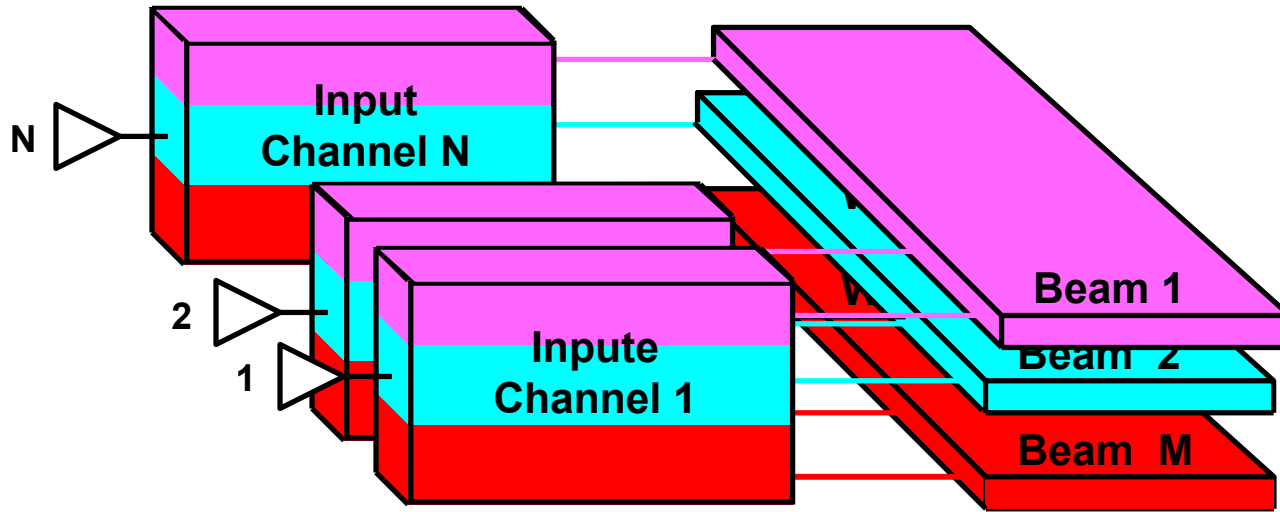
- **Parallel System Analysis**



- *Latency vs. Throughput*
- **Corner Turn**
- *Dynamic Load Balancing*

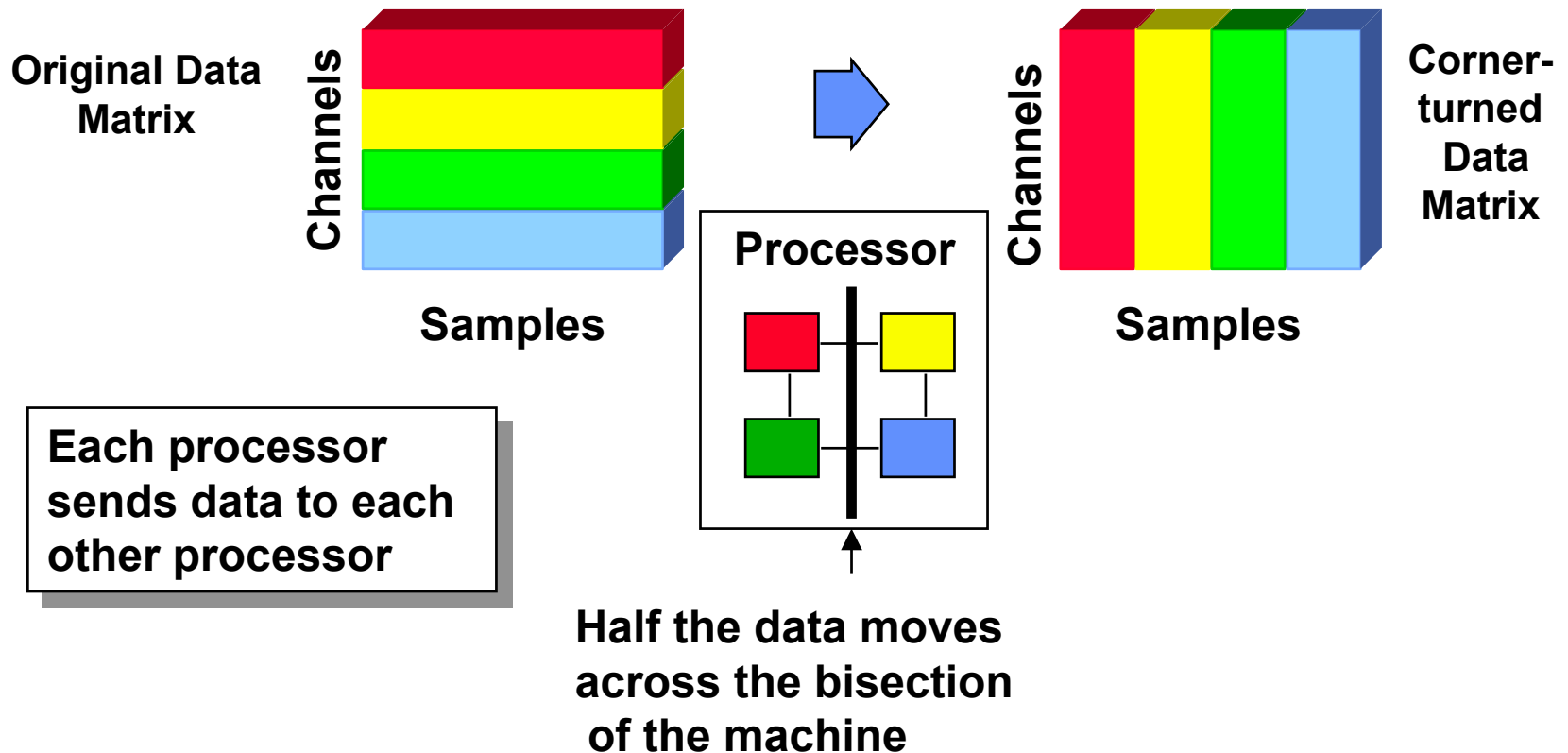
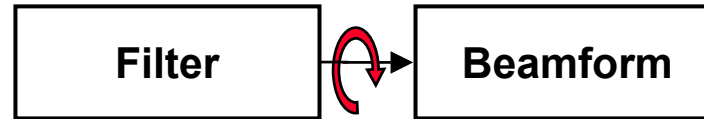
- Software Frameworks
- Summary

# Channel Space -> Beam Space



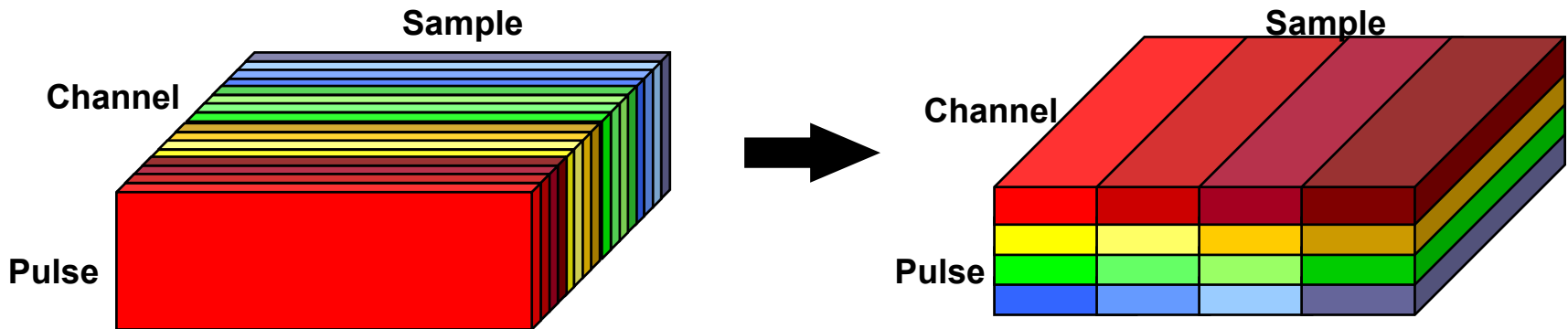
- Data enters system via different channels
- Filtering performed in a channel parallel fashion
- Beamforming requires combining data from multiple channels

# Corner Turn Operation



# Corner Turn for Signal Processing

Corner turn changes matrix distribution to exploit parallelism in successive pipeline stages



## Corner Turn Model

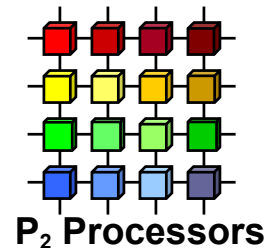
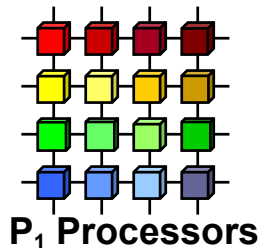
$$T_{CT} = \frac{P_1 P_2 (\alpha + B/\beta)}{Q}$$

B = Bytes per message

Q = Parallel paths

$\alpha$  = Message startup cost

$\beta$  = Link bandwidth



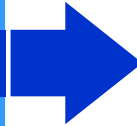
All-to-all communication where each of  $P_1$  processors sends a message of size B to each of  $P_2$  processors  
Total data cube size is  $P_1 P_2 B$

# Outline

---

- Introduction
- Processing Algorithms

- **Parallel System Analysis**

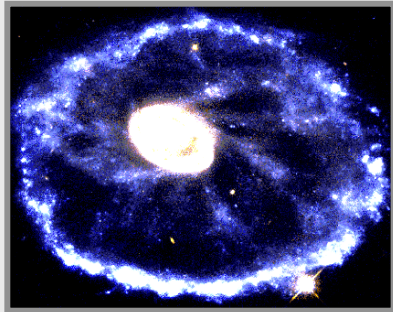


- *Latency vs. Throughput*
- *Corner Turn*
- *Dynamic Load Balancing*

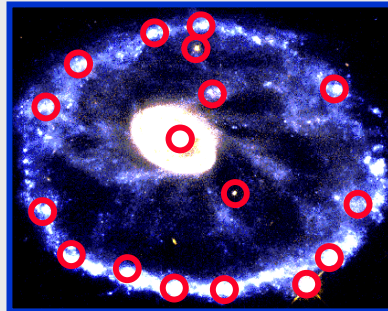
- Software Frameworks
- Summary

# Dynamic Load Balancing

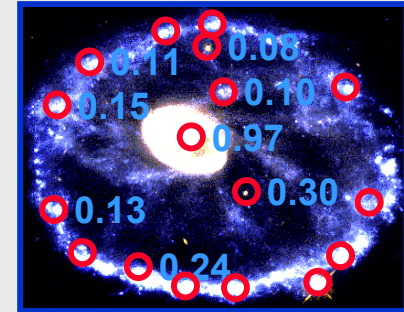
## Image Processing Pipeline



Detection



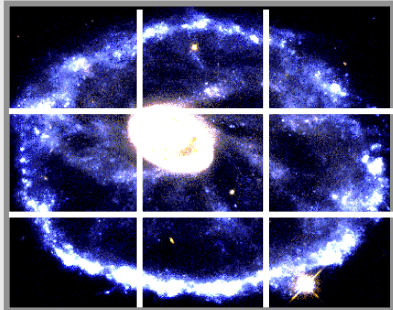
Estimation



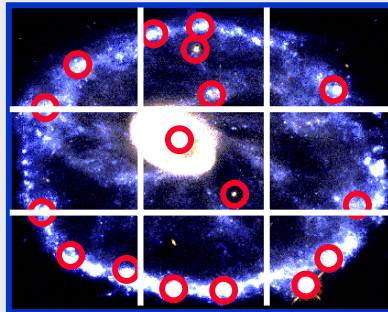
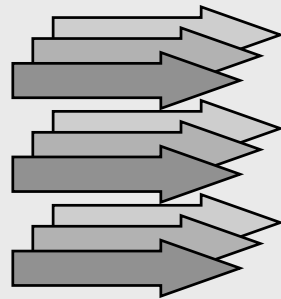
Work  $\propto$  Pixels  
(static)

Work  $\propto$  Detections  
(dynamic)

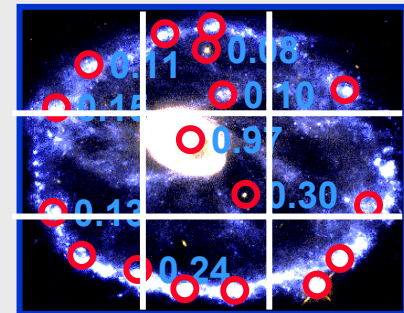
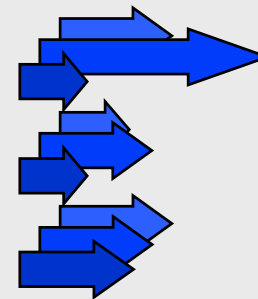
## Static Parallel Implementation



Load: balanced



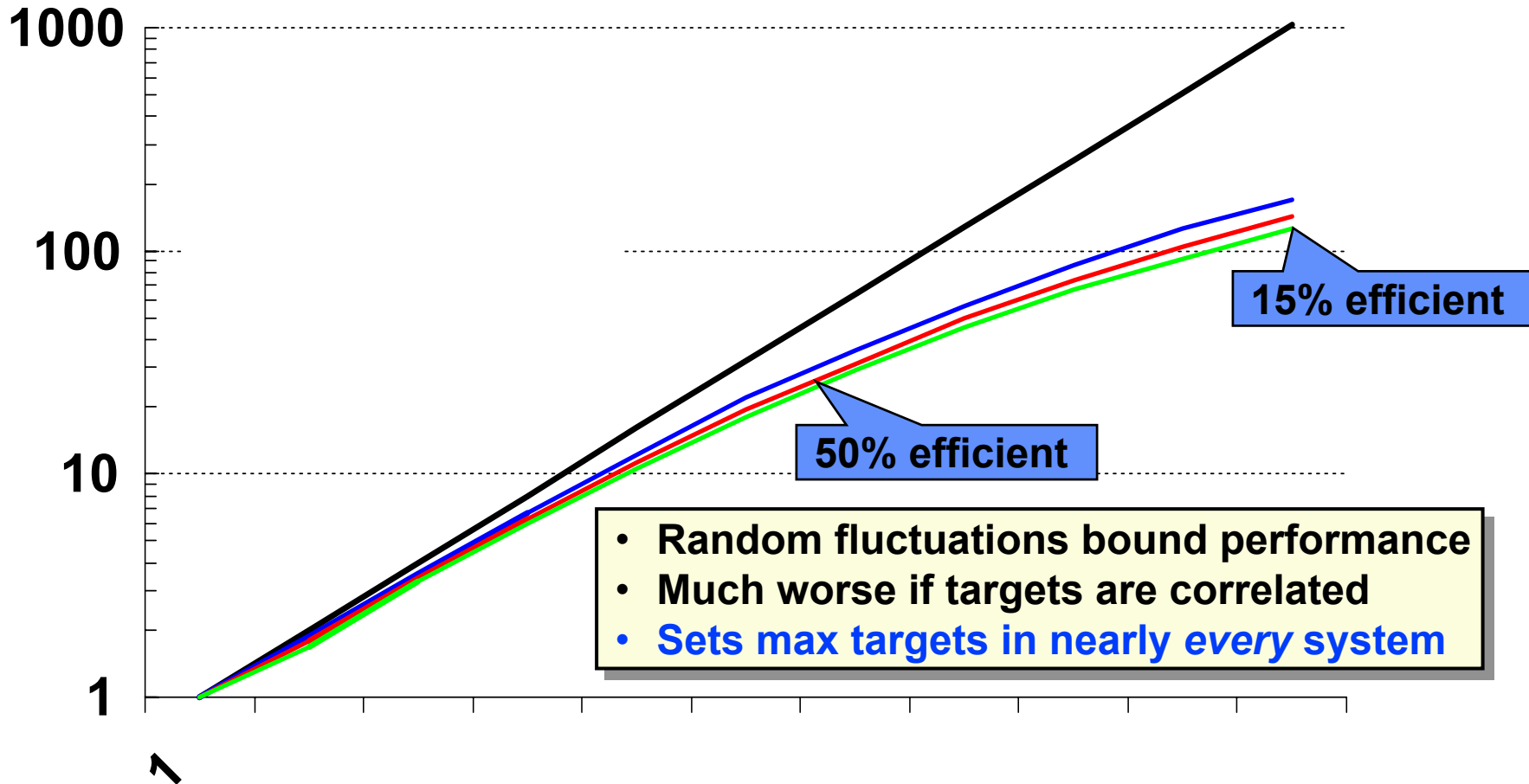
Load: unbalanced



- Static parallelism implementations lead to unbalanced loads

# Static Parallelism and Poisson's Wall

i.e. "Ball into Bins"



M = # units of work  
f = allowed failure rate

# Static Derivation

$$\text{speedup} \equiv \frac{N_d}{N_f}$$

$N_d \equiv$  Total detections

$N_f \equiv$  Allowed detections with failure rate  $f$

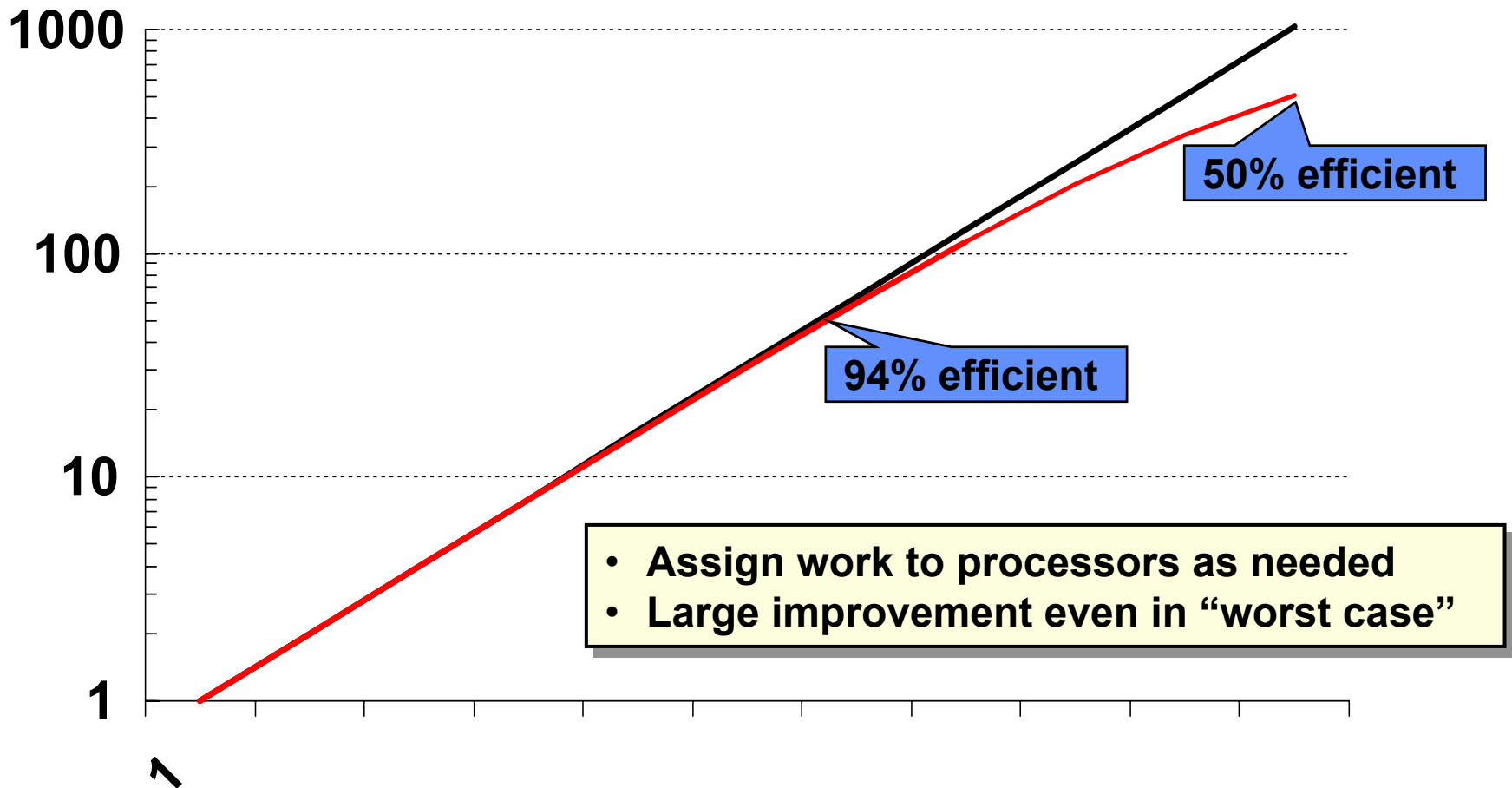
$N_p \equiv$  Number of processors

$$\lambda \equiv N_d / N_p$$

$$N_f : P_\lambda(N_f)^{N_p} = 1 - f$$

$$P_\lambda(N) = \sum_{n=0, N} \frac{\lambda^n e^{-\lambda}}{n!}$$

# Dynamic Parallelism



M = # units of work  
f = allowed failure rate

# Dynamic Derivation

$$\text{worst case speedup} = \frac{N_d}{\lambda + gN_d} = \frac{N_d}{N_d/N_p + gN_d} = \frac{N_p}{1 + gN_p}$$

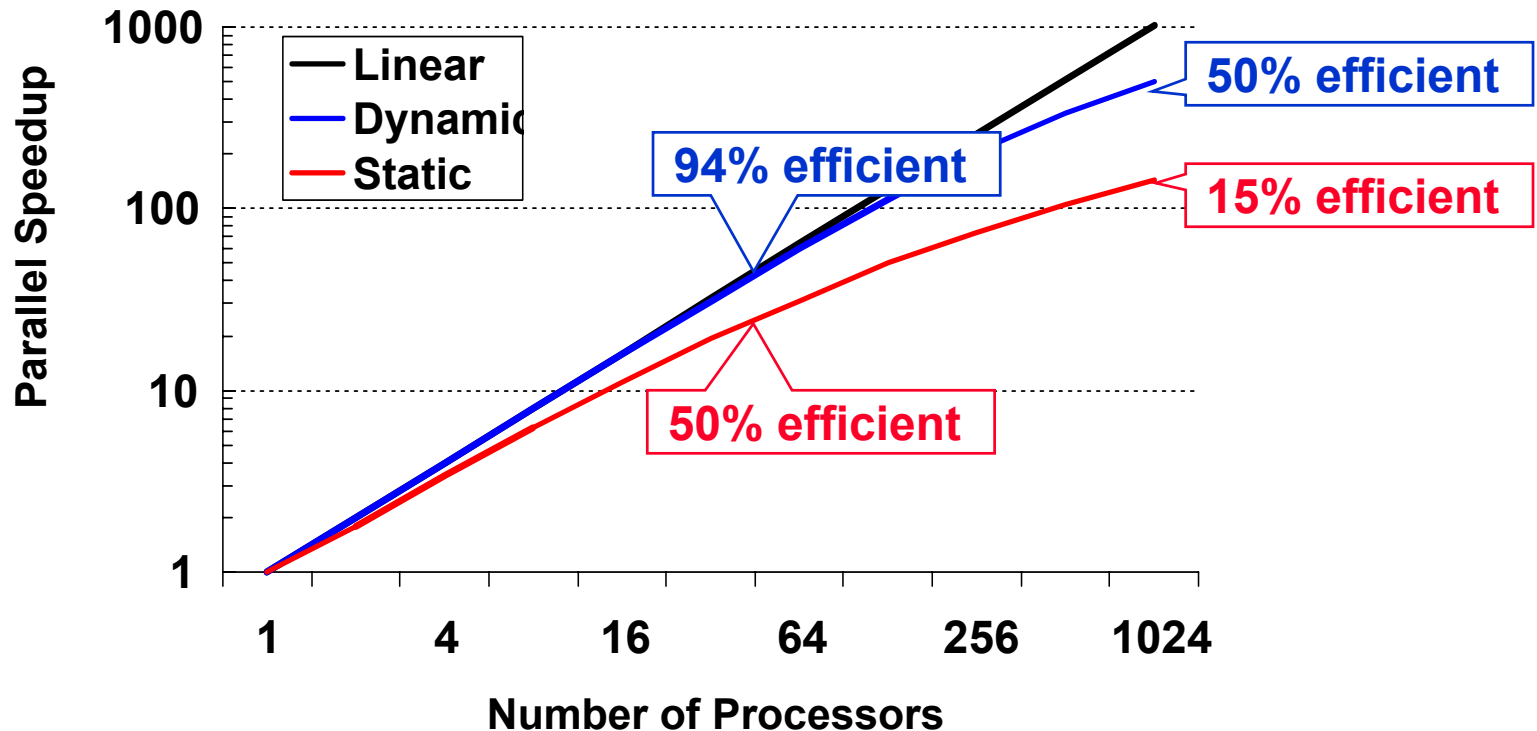
$N_d \equiv$  Total detections

$N_p \equiv$  Number of processors

$g \equiv$  granularity of work

$$\lambda \equiv N_d/N_p$$

# Static vs Dynamic Parallelism



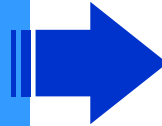
- **Dynamic** parallelism delivers good performance even in worst case
- **Static** parallelism is limited by random fluctuations (up to 85% of processors are idle)

# Outline

---

- Introduction
- Processing Algorithms
- Parallel System Analysis

- **Software Frameworks**



- *PVL*
- *PETE*
- *S3P*
- *MatlabMPI*

- Summary

# Current Standards for Parallel Coding



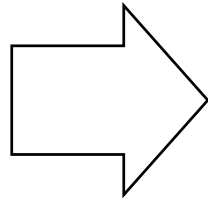
- **Industry standards (e.g. VSIPL, MPI) represent a significant improvement over coding with vendor-specific libraries**
- **Next generation of object oriented standards will provide enough support to write truly portable **scalable** applications**

# Goal: Write Once/Run Anywhere/Anysize

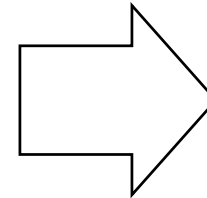
Develop code  
on a workstation

```
A = B + C;  
D = FFT(A);
```

(matlab like)



Demo Real-Time  
with a cluster  
(no code changes;  
roll-on/roll-off)



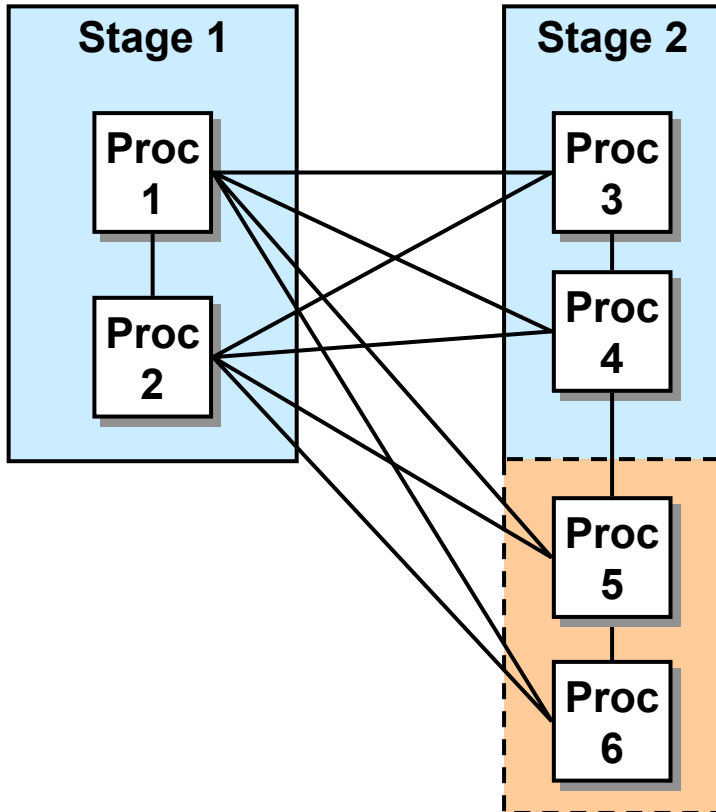
Deploy on  
Embedded System  
(no code changes)



**Scalable/portable code provides high productivity**

# Current Approach to Parallel Code

## Algorithm + Mapping



## Code

```
while(!done)
{
  if ( rank()==1 || rank()==2 )
    stage1 ();
  else if ( rank()==3 || rank()==4 )
    stage2();
}
```

```
while(!done)
{
  if ( rank()==1 || rank()==2 )
    stage1();
  else if ( rank()==3 || rank()==4) ||
    rank()==5 || rank==6 )
    stage2();
}
```

- Algorithm and hardware mapping are linked
- Resulting code is non-scalable and non-portable

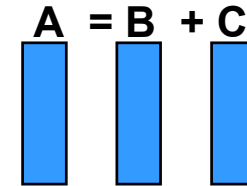
# Scalable Approach

```
#include <Vector.h>
#include <AddPvl.h>

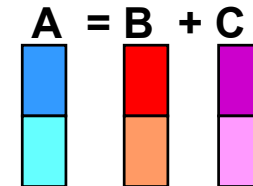
void addVectors(aMap, bMap, cMap) {
  Vector< Complex<Float> > a('a', aMap, LENGTH);
  Vector< Complex<Float> > b('b', bMap, LENGTH);
  Vector< Complex<Float> > c('c', cMap, LENGTH);

  b = 1;
  c = 2;
  a=b+c;
}
```

## Single Processor Mapping

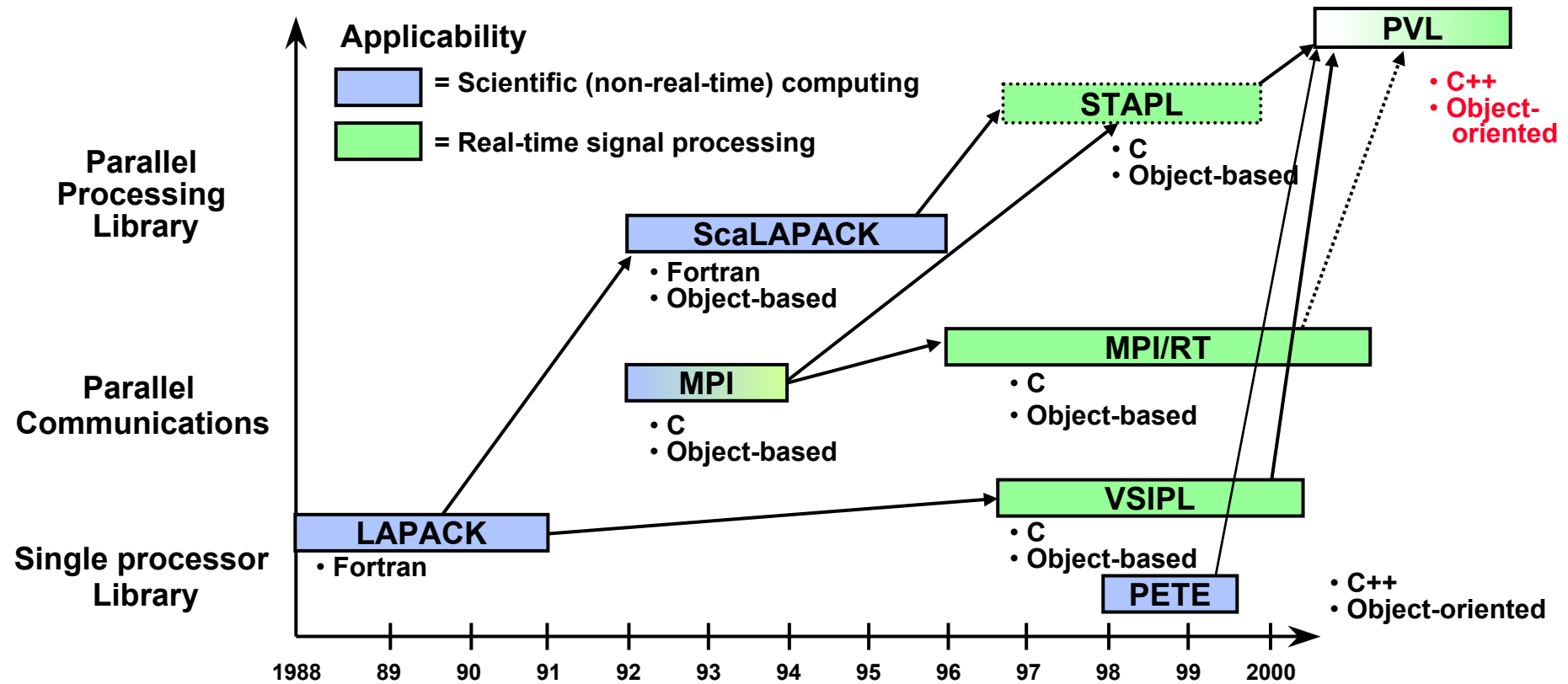


## Multi Processor Mapping



- Single processor and multi-processor code are the *same*
- *Maps* can be changed without changing software
- High level code is compact

# PVL Evolution



- Transition technology from scientific computing to real-time
- Moving from procedural (Fortran) to object oriented (C++)

# Anatomy of a PVL Map

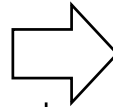
Vector/Matrix



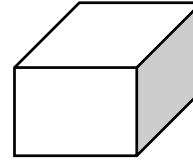
Computation



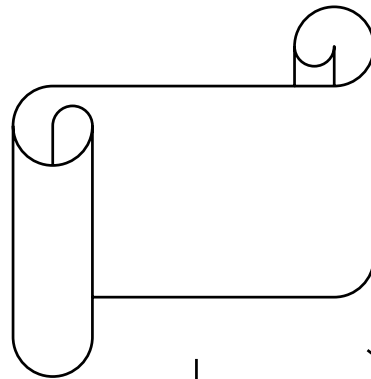
Conduit



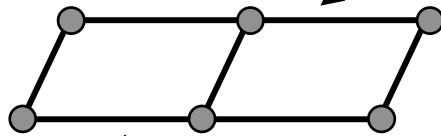
Task



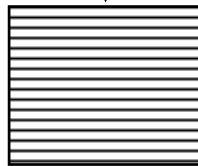
Map



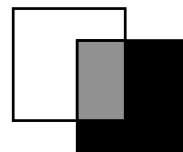
Grid



Distribution





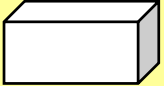

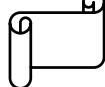
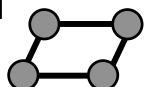
Overlap



$\{0,2,4,6,8,10\}$   
List of Nodes

- All PVL objects contain maps
- PVL Maps contain
  - Grid
  - List of nodes
  - Distribution
  - Overlap

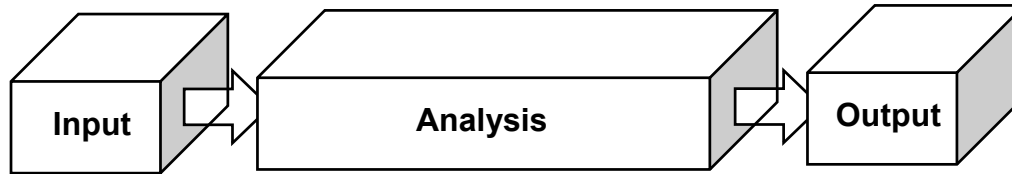
# Library Components

	<u>Class</u>	<u>Description</u>	<u>Parallelism</u>
<b>Signal Processing &amp; Control</b>	<b>Vector/Matrix</b> 	Used to perform matrix/vector algebra on data spanning multiple processors	Data
	<b>Computation</b> 	Performs signal/image processing functions on matrices/vectors (e.g. FFT, FIR, QR)	Data & Task
	<b>Task</b> 	Supports algorithm decomposition (i.e. the boxes in a signal flow diagram)	Task & Pipeline
	<b>Conduit</b> 	Supports data movement between tasks (i.e. the arrows on a signal flow diagram)	Task & Pipeline
<b>Mapping</b>	<b>Map</b> 	Specifies how Tasks, Vectors/Matrices, and Computations are distributed on processor	Data, Task & Pipeline
	<b>Grid</b> 	Organizes processors into a 2D layout	

- Simple mappable components support data, task and pipeline parallelism

# PVL Layered Architecture

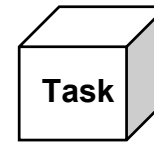
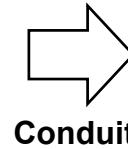
**Application**



Productivity



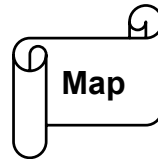
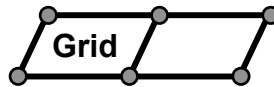
Comp



User Interface

**Parallel Vector Library**

Performance



Portability

Math Kernel (VSIPL)

Messaging Kernel (MPI)

Hardware Interface

**Hardware**



Workstation



Intel Cluster



PowerPC Cluster



Embedded Board



Embedded Multi-computer

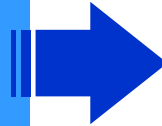
- Layers enable simple interfaces between the application, the library, and the hardware

# Outline

---

- Introduction
- Processing Algorithms
- Parallel System Analysis

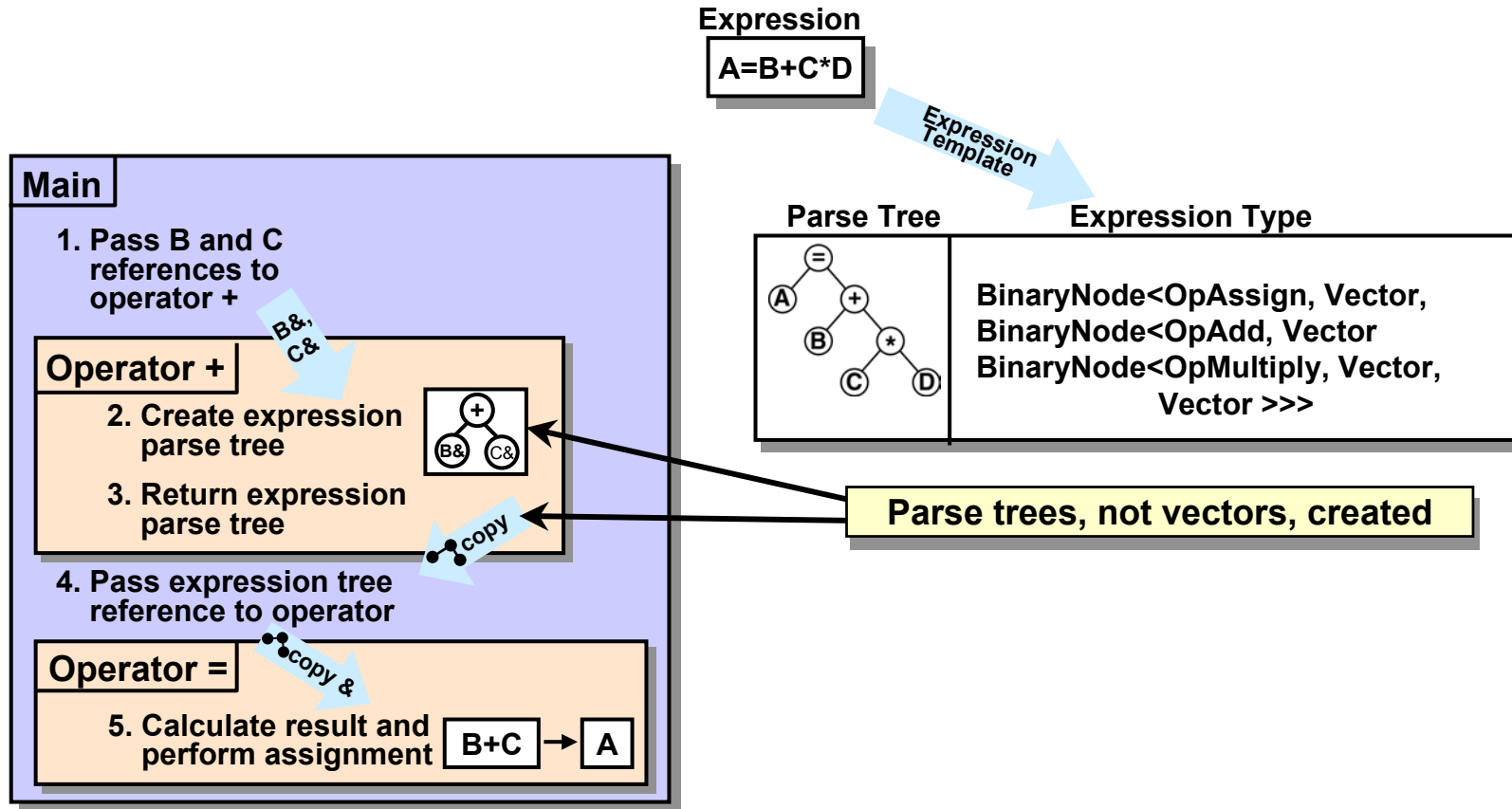
- **Software Frameworks**



- *PVL*
- *PETE*
- *S3P*
- *MatlabMPI*

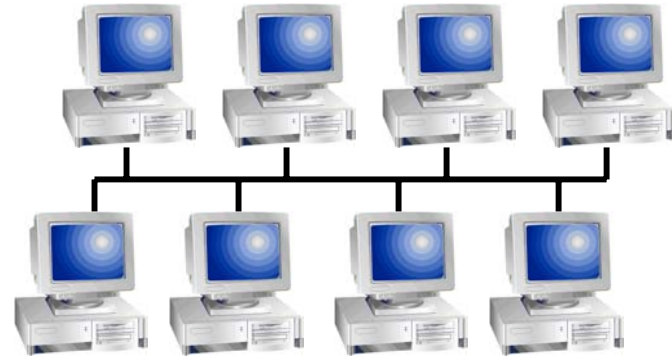
- Summary

# C++ Expression Templates and PETE



- Expression Templates enhance performance by allowing temporary variables to be avoided

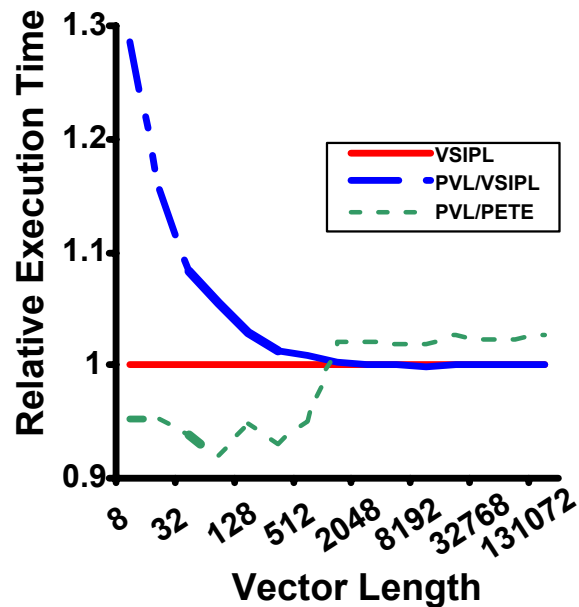
# Experimental Platform



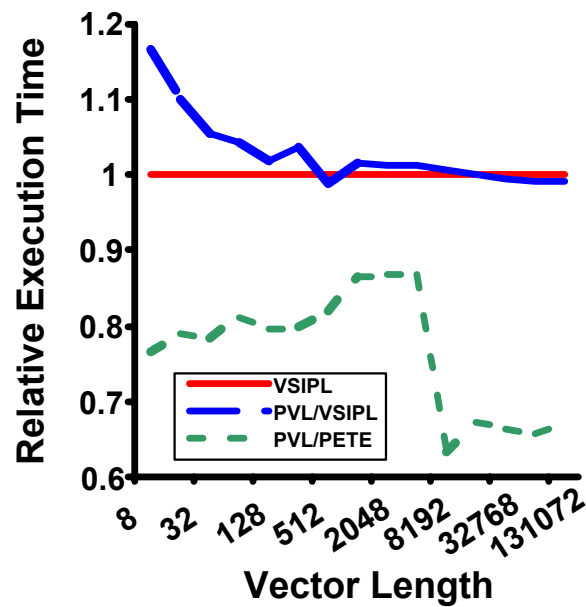
- **Network of 8 Linux workstations**
  - 800 MHz Pentium III processors
- **Communication**
  - Gigabit ethernet, 8-port switch
  - Isolated network
- **Software**
  - Linux kernel release 2.2.14
  - GNU C++ Compiler
  - MPICH communication library over TCP/IP

# Experiment 1: Single Processor

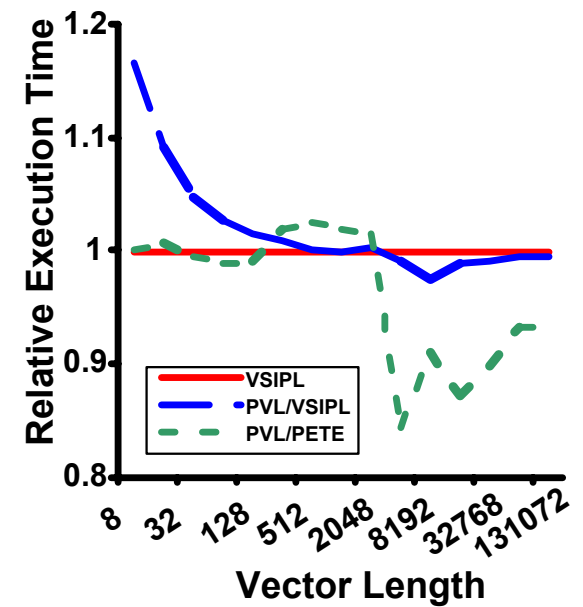
A=B+C



A=B+C\*D



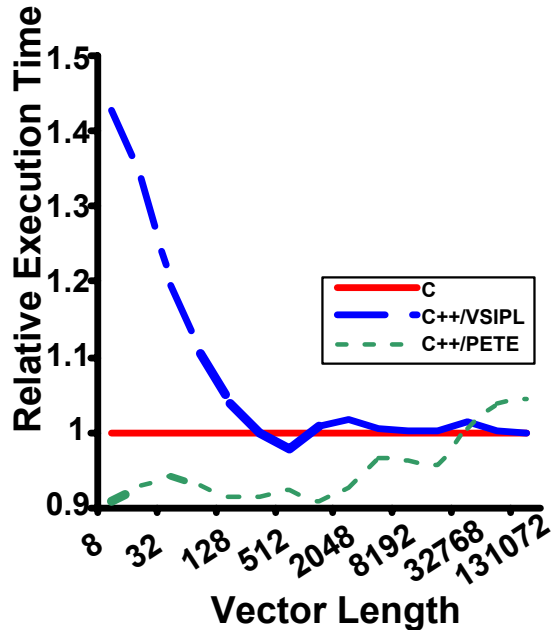
A=B+C\*D/E+fft(F)



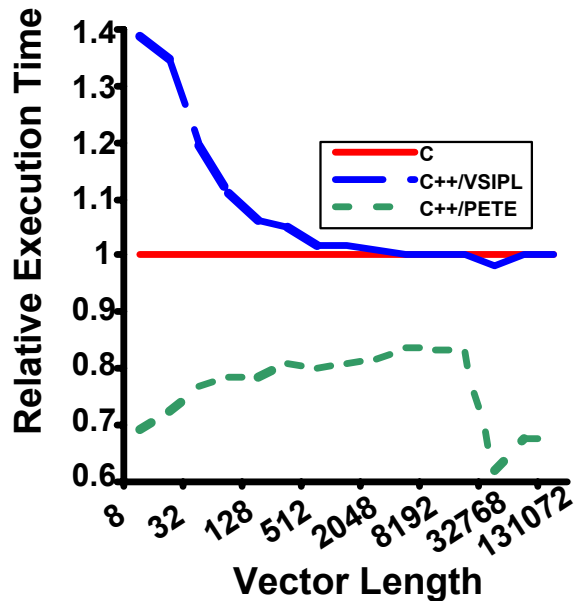
- PVL with VSIBL has a small overhead
- PVL with PETE can surpass VSIBL

# Experiment 2: Multi-Processor (simple communication)

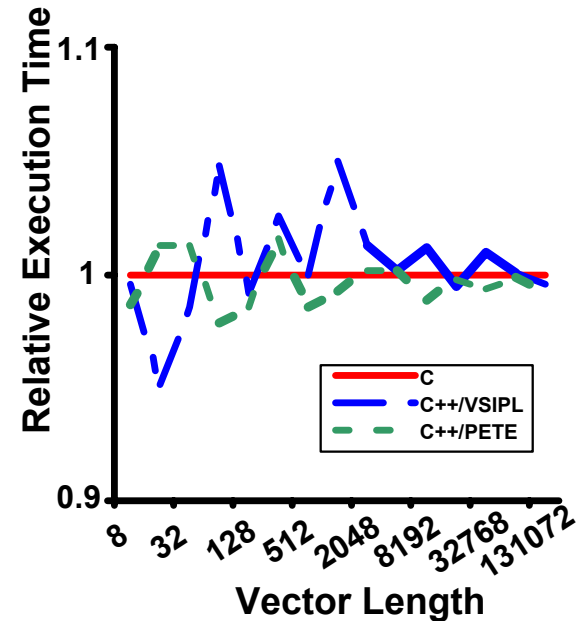
A=B+C



A=B+C\*D

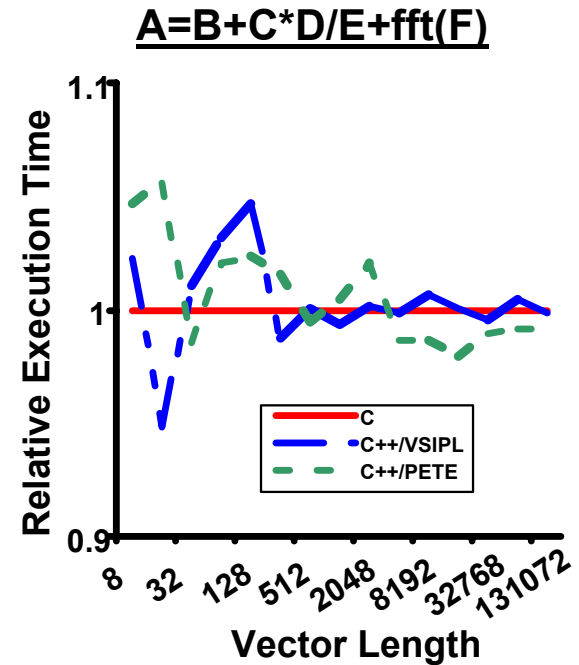
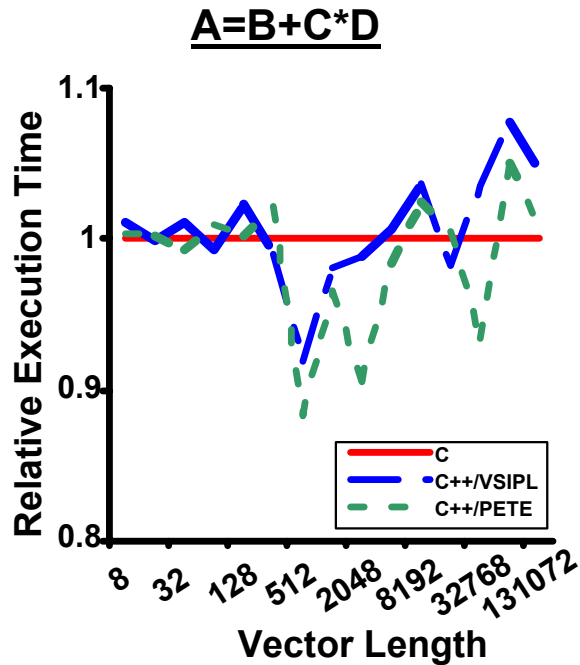
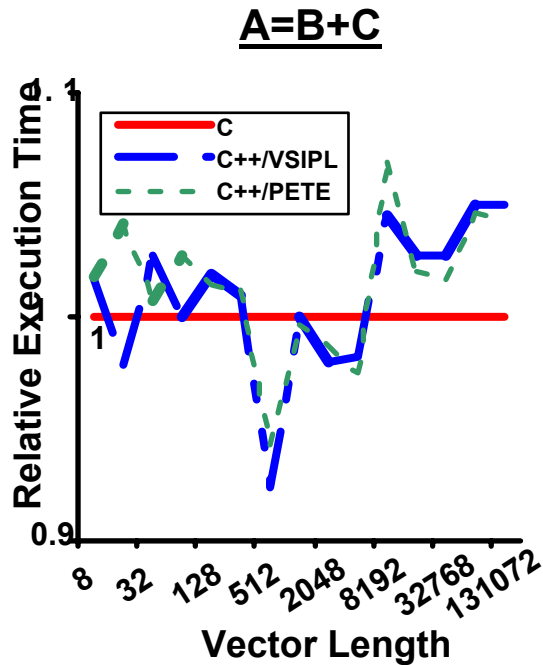


A=B+C\*D/E+fft(F)



- PVL with VSIP has a small overhead
- PVL with PETE can surpass VSIP

# Experiment 3: Multi-Processor (complex communication)



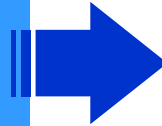
- Communication dominates performance

# Outline

---

- Introduction
- Processing Algorithms
- Parallel System Analysis

- **Software Frameworks**

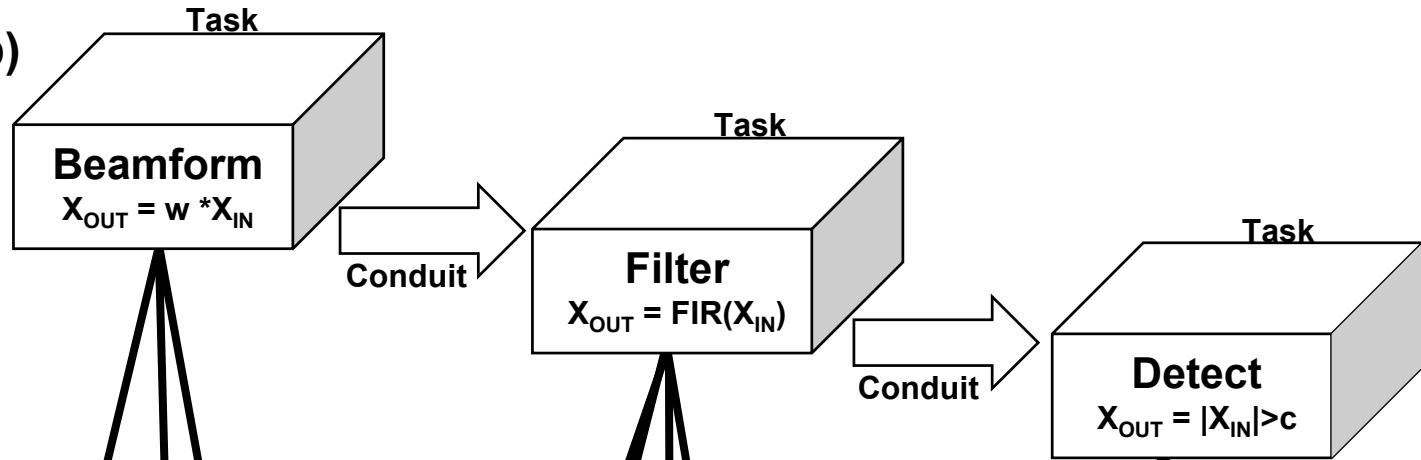


- *PVL*
- *PETE*
- ***S3P***
- *MatlabMPI*

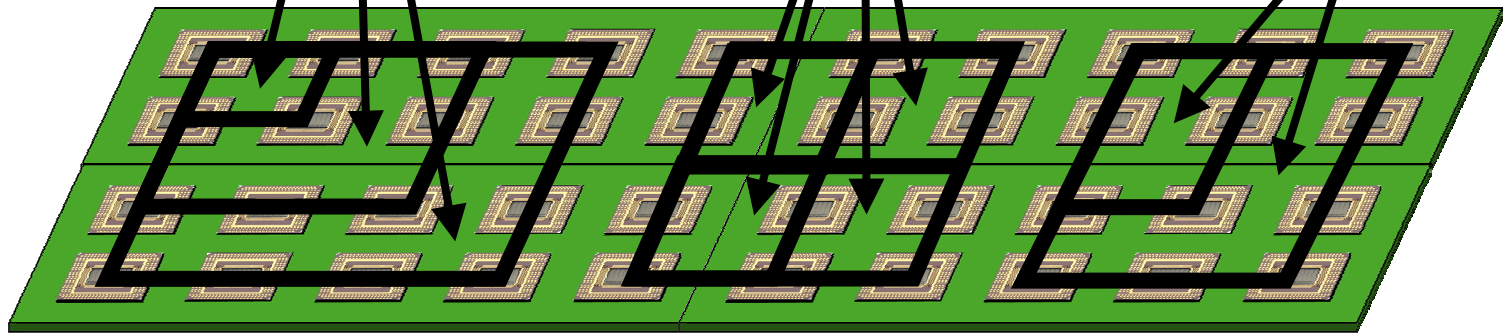
- Summary

# S3P Framework Requirements

Decomposable  
into Tasks (comp)  
and Conduits  
(comm)



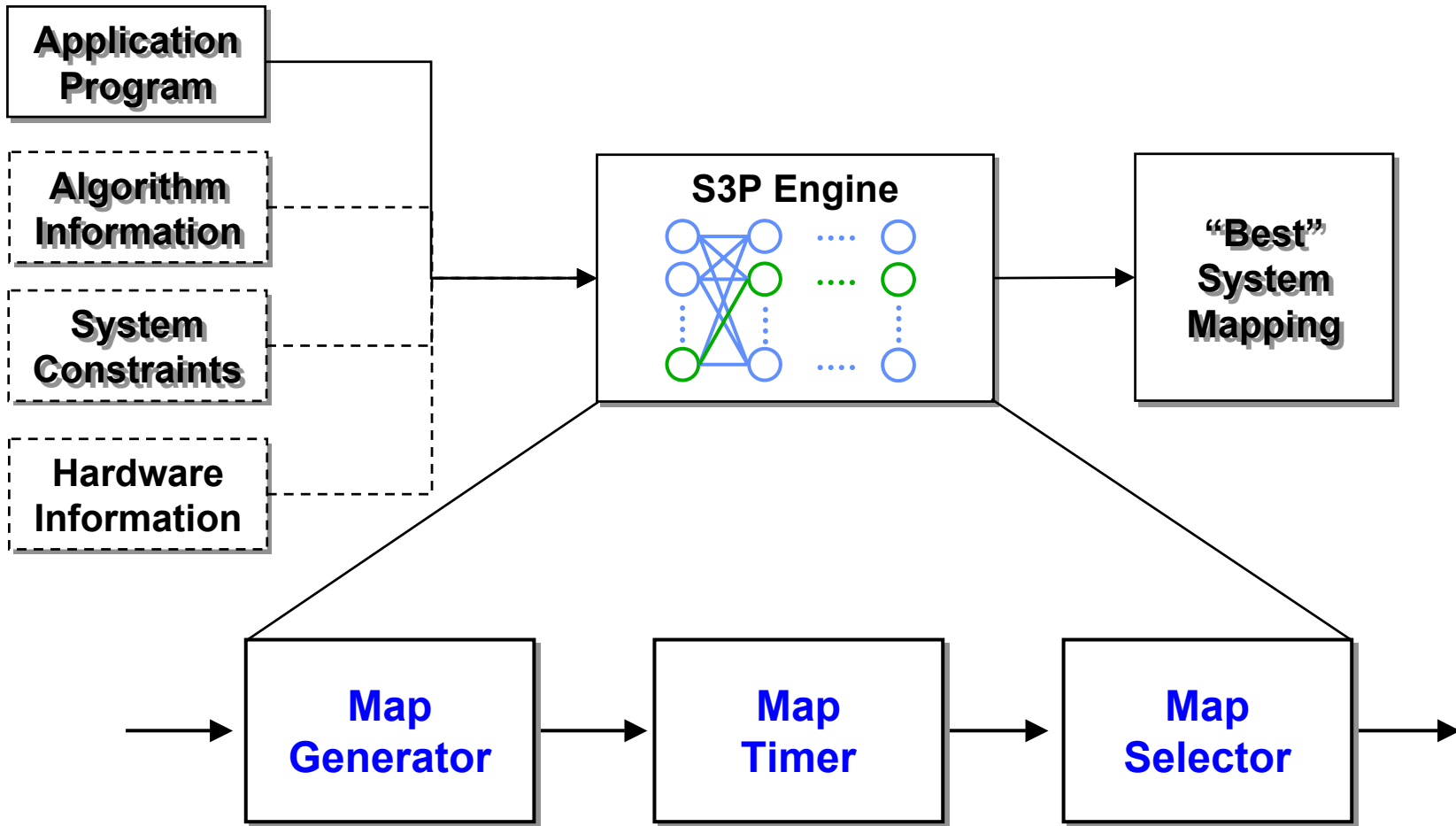
Mappable  
to different sets  
of hardware



Measurable  
resource usage of  
each mapping

- Each compute stage can be mapped to different sets of hardware and timed

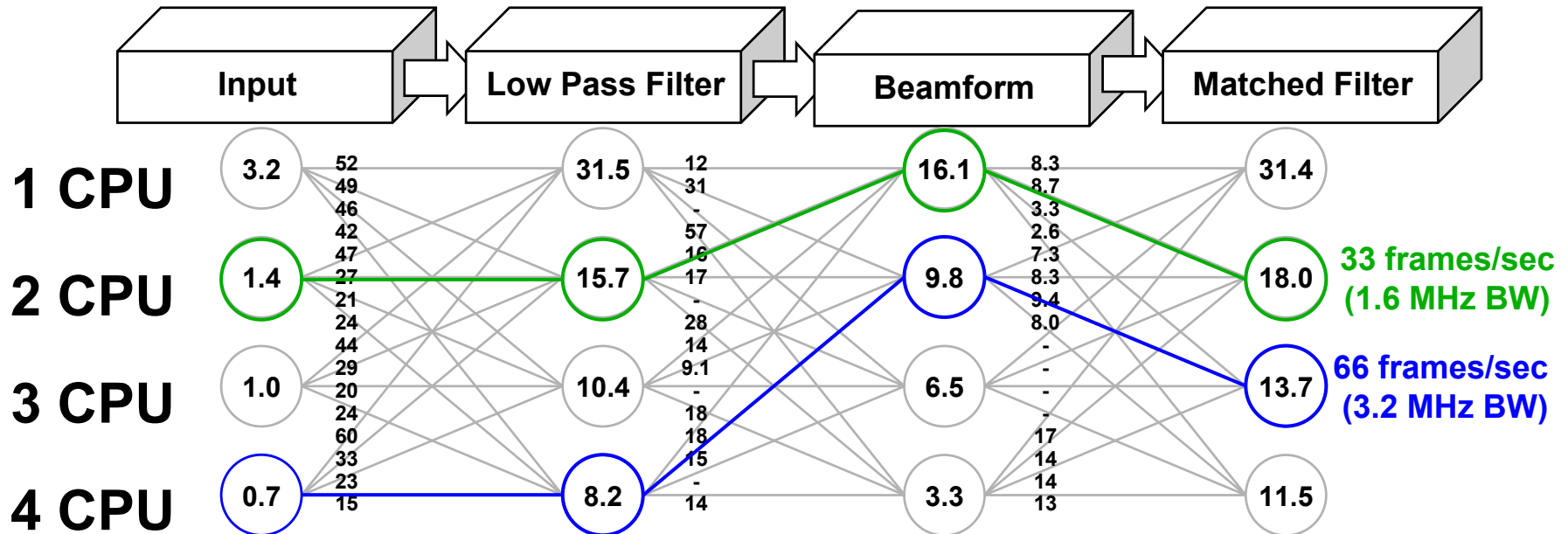
# S3P Engine



- **Map Generator** constructs the system graph for all candidate mappings
- **Map Timer** times each node and edge of the system graph
- **Map Selector** searches the system graph for the optimal set of maps

# Test Case: Min(#CPU | Throughput)

- Vary number of processors used on each stage
- Time each computation stage and communication conduit
- Find path with minimum bottleneck



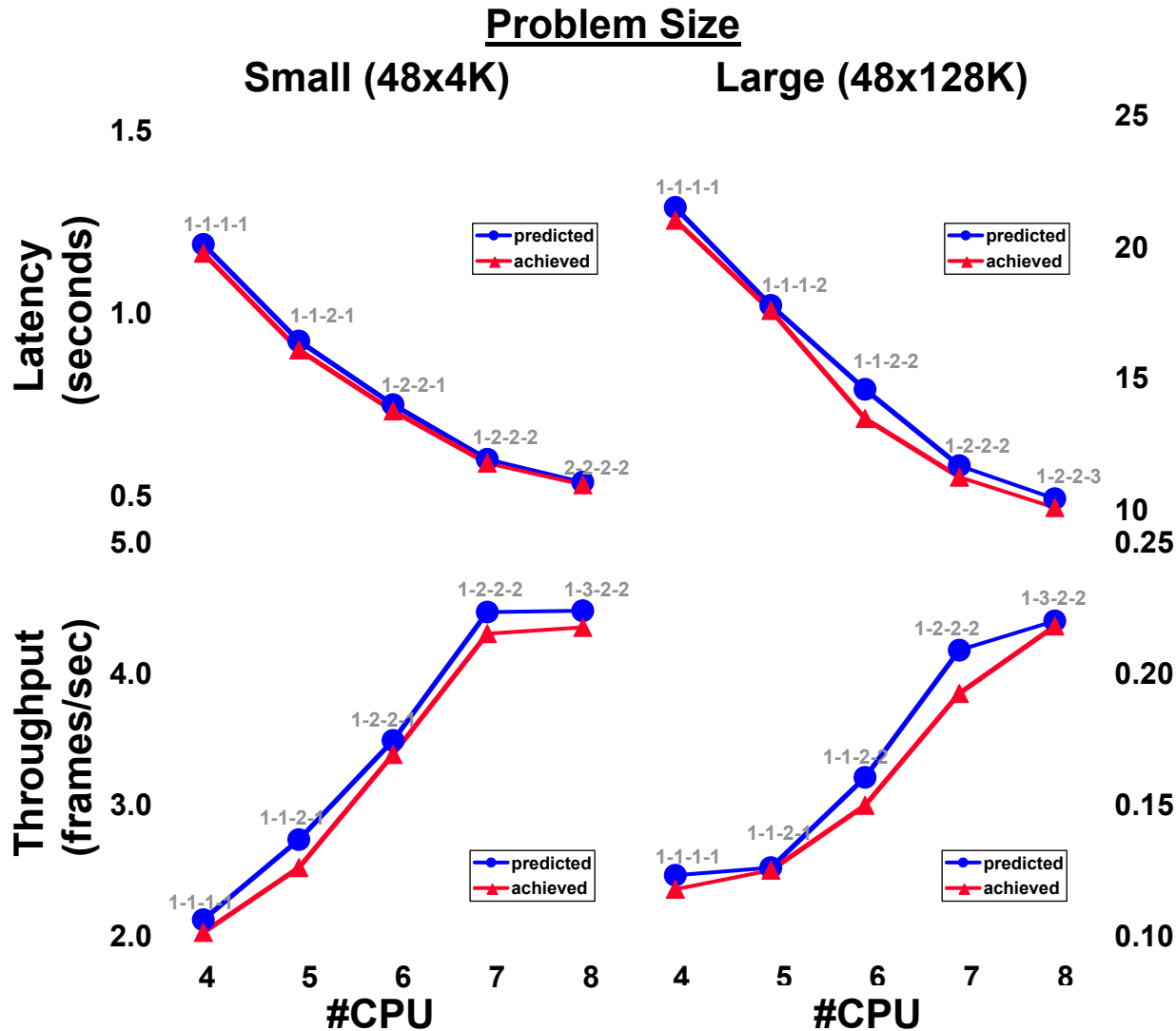
# Dynamic Programming

- Graph construct is very general
- Widely used for optimization problems
- Many efficient techniques for choosing “best” path (under constraints) such as Dynamic Programming

```
N = total hardware units
M = number of tasks
Pi = number of mappings for task i

t = M
pathTable[M][N] = all infinite weight paths
for( j:1..M ){ for( k:1..Pj ){ for( i:j+1..N-t+1 ){
  if( i-size[k] >= j ){
    if( j > 1 ){
      w = weight[pathTable[j-1][i-size[k]]] +
        weight[k] +
        weight[edge[last[pathTable[j-1][i-size[k]]],k]
      p = addVertex[pathTable[j-1][i-size[k]], k]
    }else{
      w = weight[k]
      p = makePath[k]
    }
    if( weight[pathTable[j][i]] > w ){
      pathTable[j][i] = p
    }
  }
}
}
}
t = t - 1
}
```

# Predicted and Achieved Latency and Throughput



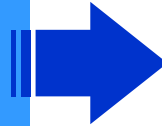
- **Find**
  - Min(latency | #CPU)
  - Max(throughput | #CPU)
- **S3P selects correct optimal mapping**
- **Excellent agreement between S3P predicted and achieved latencies and throughputs**

# Outline

---

- Introduction
- Processing Algorithms
- Parallel System Analysis

- **Software Frameworks**

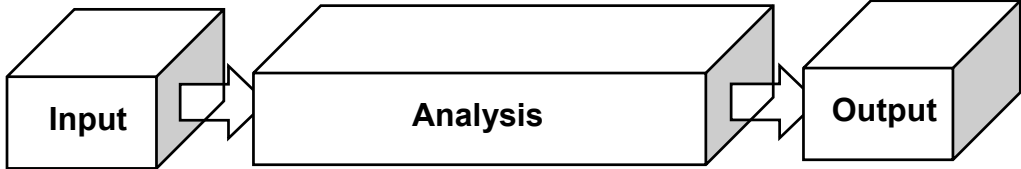


- *PVL*
- *PETE*
- *S3P*
- ***MatlabMPI***

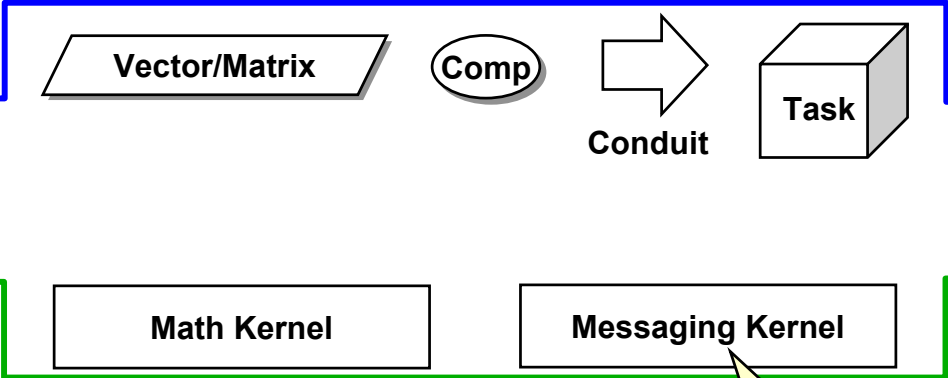
- Summary

# Modern Parallel Software Layers

Application



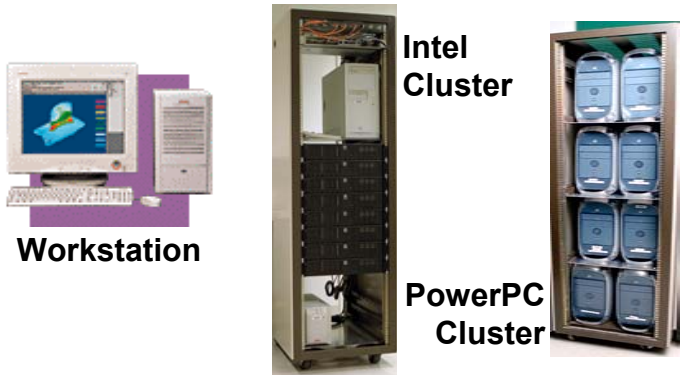
Parallel Library



User Interface

Hardware Interface

Hardware



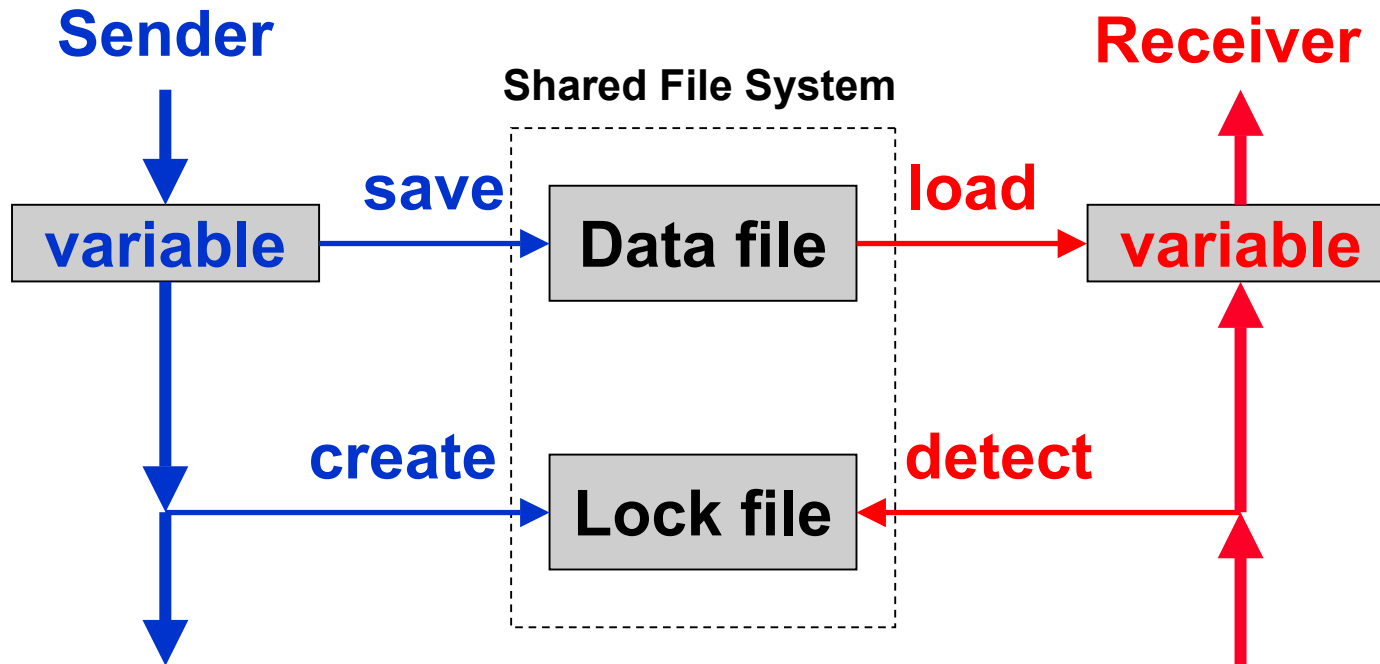
- Can build any parallel application/library on top of a few basic messaging capabilities
- MatlabMPI provides this Messaging Kernel

# MatlabMPI “Core Lite”

- **Parallel computing requires eight capabilities**
  - **MPI\_Run** launches a Matlab script on multiple processors
  - **MPI\_Comm\_size** returns the number of processors
  - **MPI\_Comm\_rank** returns the id of each processor
  - **MPI\_Send** sends Matlab variable(s) to another processor
  - **MPI\_Recv** receives Matlab variable(s) from another processor
  - **MPI\_Init** called at beginning of program
  - **MPI\_Finalize** called at end of program

# MatlabMPI: Point-to-point Communication

`MPI_Send (dest, tag, comm, variable);`



`variable = MPI_Recv (source, tag, comm);`

- **Sender** saves variable in Data file, then creates Lock file
- **Receiver** detects Lock file, then loads Data file

# Example: Basic Send and Receive

- Initialize
- Get processor ranks

- Execute send
- Execute receive

- Finalize
- Exit

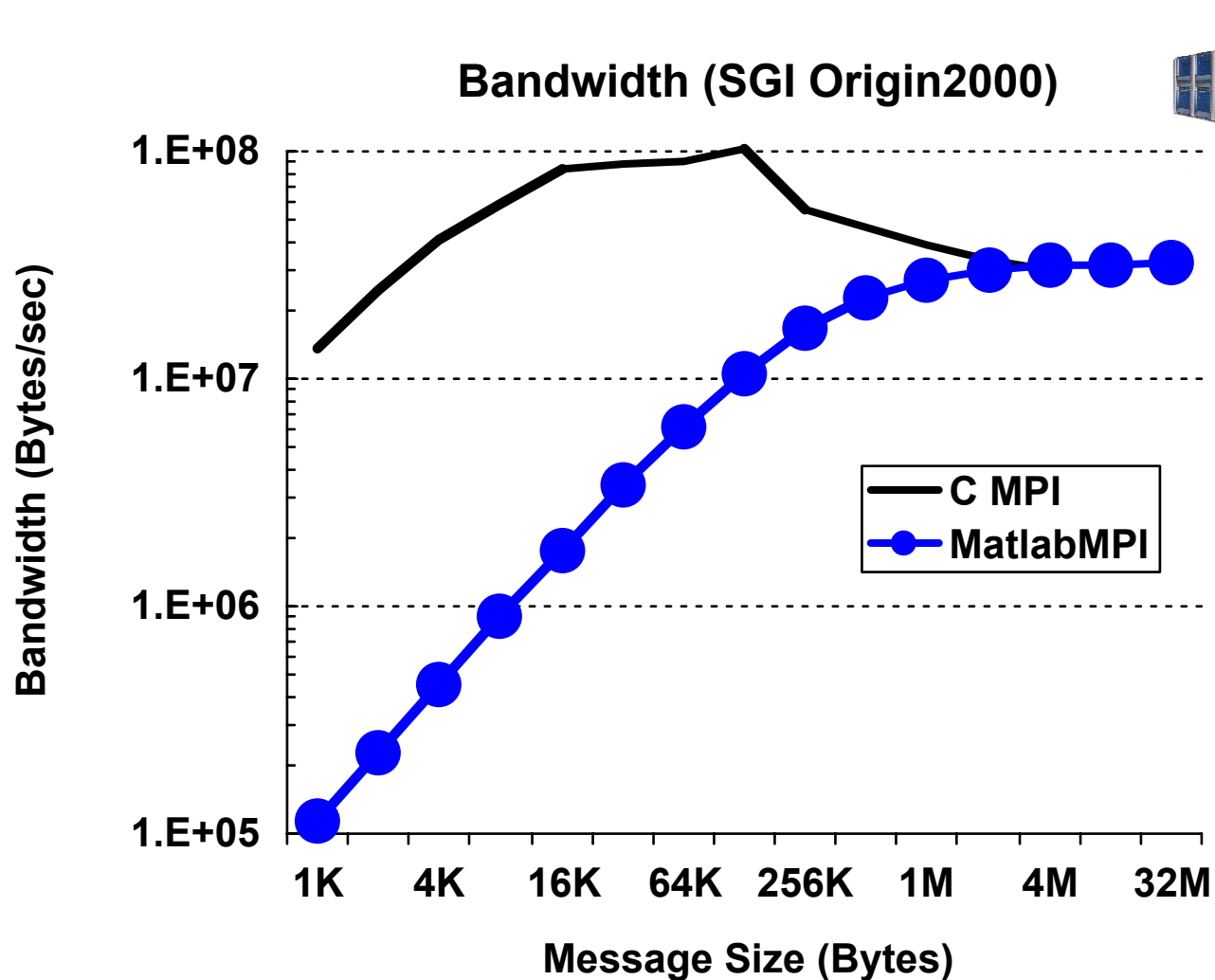
```
MPI_Init; % Initialize MPI.
comm = MPI_COMM_WORLD; % Create communicator.
comm_size = MPI_Comm_size(comm); % Get size.
my_rank = MPI_Comm_rank(comm); % Get rank.
source = 0; % Set source.
dest = 1; % Set destination.
tag = 1; % Set message tag.

if(comm_size == 2) % Check size.
    if(my_rank == source) % If source.
        data = 1:10; % Create data.
        MPI_Send(dest,tag,comm,data); % Send data.
    end
    if(my_rank == dest) % If destination.
        data=MPI_Recv(source,tag,comm); % Receive data.
    end
end

MPI_Finalize; % Finalize Matlab MPI.
exit; % Exit Matlab
```

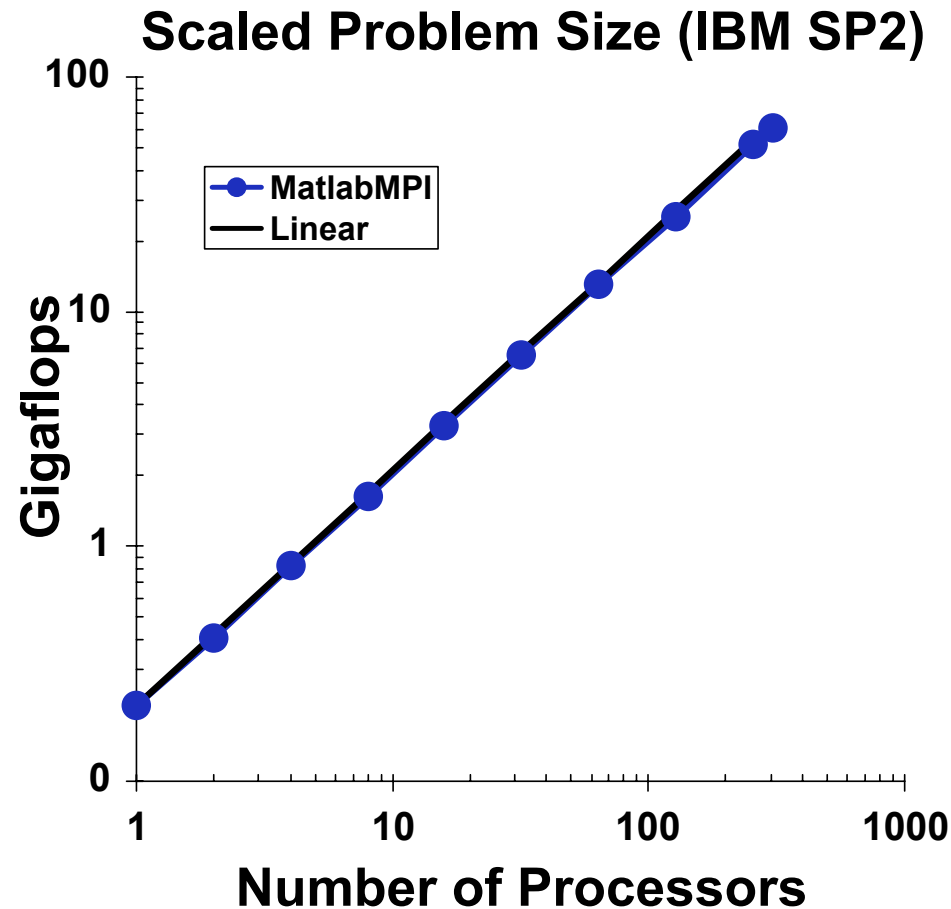
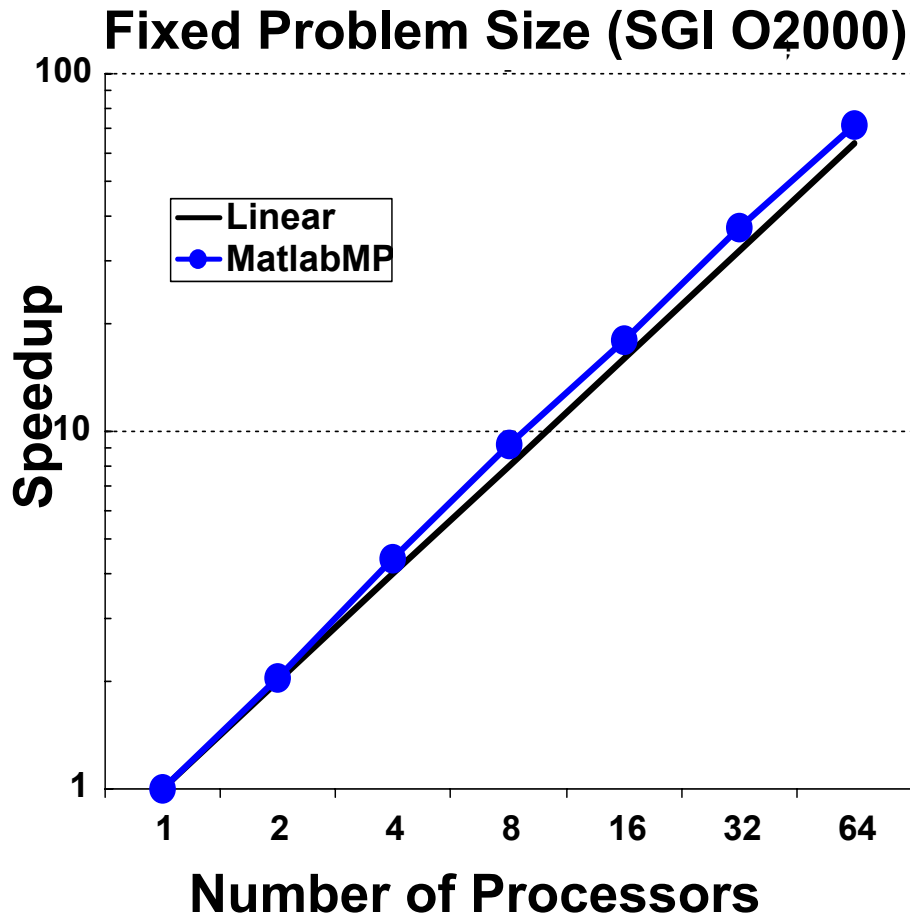
- Uses standard message passing techniques
- Will run anywhere Matlab runs
- Only requires a common file system

# MatlabMPI vs MPI bandwidth



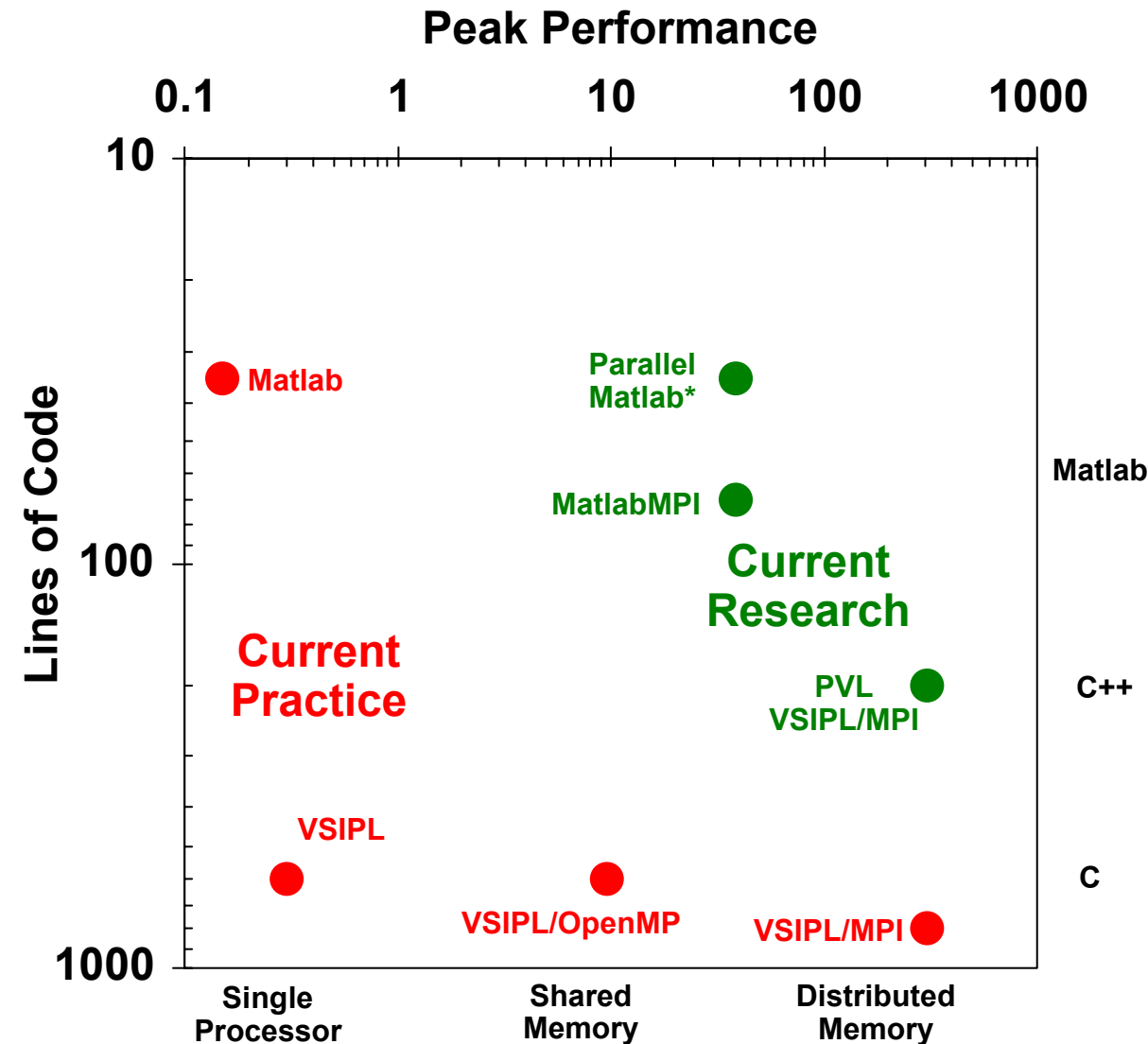
- Bandwidth matches native C MPI at large message size
- Primary difference is latency (35 milliseconds vs. 30 microseconds)

# Image Filtering Parallel Performance



- Achieved “classic” super-linear speedup on fixed problem
- Achieved speedup of ~300 on 304 processors on scaled problem

# Productivity vs. Performance



- Programmed image filtering several ways
  - Matlab
  - VSIPL
  - VSIPL/OpenMPI
  - VSIPL/MPI
  - PVL
  - MatlabMPI
- MatlabMPI provides
  - high productivity
  - high performance

# Summary

- **Exploiting parallel processing for streaming applications presents unique software challenges.**
- **The community is developing software libraries to address many of these challenges:**
  - **Exploits C++ to easily express data/task parallelism**
  - **Separates parallel hardware dependencies from software**
  - **Allows a variety of strategies for implementing dynamic applications (e.g. for fault tolerance)**
  - **Delivers high performance execution comparable to or better than standard approaches**
- **Our future efforts will focus on adding to and exploiting the features of this technology to:**
  - **Exploit dynamic parallelism**
  - **Integrate high performance parallel software underneath mainstream programming environments (e.g Matlab, IDL, ...)**
  - **Use self-optimizing techniques to maintain performance**