Parallel Matlab programming using Distributed Arrays

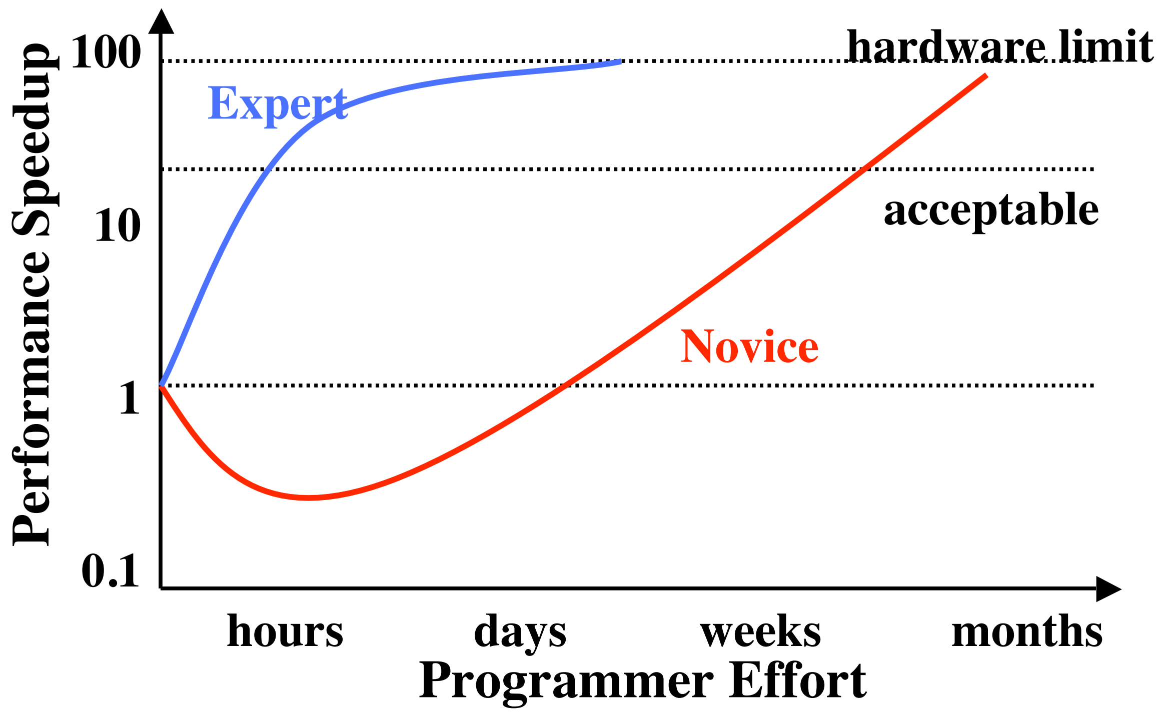
Jeremy Kepner

MIT Lincoln Laboratory

This work is sponsored by the Department of Defense under Air Force Contract FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the author and are not necessarily endorsed by the United States Government.

Goal: Think Matrices not Messages

- **In the past, writing well performing parallel programs has required a lot of code and a lot of expertise**
- **pMatlab distributed arrays eliminates the coding burden**
  - **However, making programs run fast still requires expertise**
- **This talk illustrates the key math concepts experts use to make parallel programs perform well**



Outline

- **Parallel Design**
- Distributed Arrays
- Concurrency vs Locality
- Execution
- Summary

- ***Serial Program***
- ***Parallel Execution***
- ***Distributed Arrays***
- ***Explicitly Local***

Serial Program

Math	Matlab
$\mathbf{X},\mathbf{Y} : \mathbb{R}^{\mathrm{NxN}}$	`X = zeros(N,N);` `Y = zeros(N,N);`
$\mathbf{Y} = \mathbf{X} + 1$	`Y(:,:) = X + 1;`

- **Matlab is a high level language**
- **Allows mathematical expressions to be written concisely**
- **Multi-dimensional arrays are *fundamental* to Matlab**

Parallel Execution

Math

$P_{ID}=N_P-1$

$P_{ID}=1$

$P_{ID}=0$

$$\mathbf{X},\mathbf{Y} : \mathbb{R}^{N\mathrm{x}N}$$

$$\mathbf{Y} = \mathbf{X} + 1$$

pMatlab

Pid=Np-1

Pid=1

Pid=0

```
X = zeros(N,N);
Y = zeros(N,N);

Y(:,:) = X + 1;
```

- **Run N_P (or `Np`) copies of same program**
  - **Single Program Multiple Data (SPMD)**
- **Each copy has a unique P_{ID} (or `Pid`)**
- **Every array is *replicated* on each copy of the program**

Distributed Array Program

Math

$P_{ID}=N_P-1$

$P_{ID}=1$

$P_{ID}=0$

$$\mathbf{X},\mathbf{Y} : \mathbb{R}^{P(N) \mathrm{x} N}$$

$$\mathbf{Y} = \mathbf{X} + 1$$

pMatlab

Pid=Np-1

Pid=1

Pid=0

```
XYmap = map([Np 1],{},0:Np-1);
X = zeros(N,N,XYmap);
Y = zeros(N,N,XYmap);


Y(:,:) = X + 1;
```

- **Use P() notation (or `map`) to make a distributed array**
- **Tells program which dimension to distribute data**
- **Each program implicitly operates on only its own data (owner computes rule)**

Explicitly Local Program

Math	**pMatlab**

$\mathbf{X},\mathbf{Y} : \mathbb{R}^{P(N)xN}$	`XYmap = map([Np 1],{},0:Np-1);`
	`Xloc = local(zeros(N,N,XYmap));`
	`Yloc = local(zeros(N,N,XYmap));`

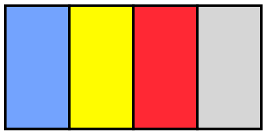
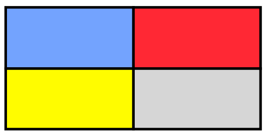
$\mathbf{Y}.\mathrm{loc} = \mathbf{X}.\mathrm{loc} + 1$	`Yloc(:,:) = Xloc + 1;`

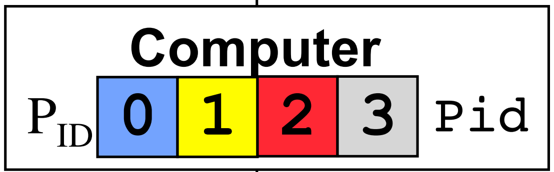
- **Use** .loc **notation (or** `local` **function) to explicitly retrieve local part of a distributed array**
- **Operation is the same as serial program, but with different data on each processor (recommended approach)**

Outline

- Parallel Design
- **Distributed Arrays**
  - *Maps*
  - *Redistribution*
- Concurrency vs Locality
- Execution
- Summary

Parallel Data Maps

Array	Math	Matlab
	$\mathbb{R}^{P(N)xN}$	`Xmap=map([Np 1],{},0:Np-1)`
	$\mathbb{R}^{NxP(N)}$	`Xmap=map([1 Np],{},0:Np-1)`
	$\mathbb{R}^{P(N)xP(N)}$	`Xmap=map([Np/2 2],{},0:Np-1)`



- **A map is a mapping of array indices to processors**
- **Can be block, cyclic, block-cyclic, or block w/overlap**
- **Use $P()$ notation (or `map`) to set which dimension to split among processors**

Maps and Distributed Arrays

A processor *map* for a numerical array is an *assignment of blocks of data to processing elements*.

```
Amap = map([Np 1],{},0:Np-1);
```

Processor Grid

Distribution
{}=default=block

List of processors

```
A = zeros(4,6,Amap);
```

P0
P1
P2
P3

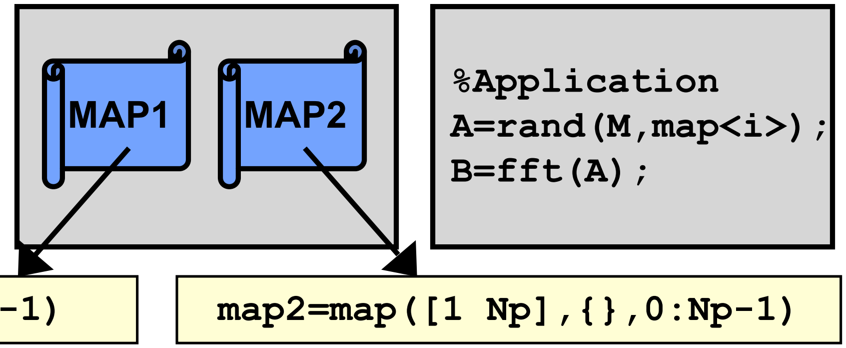
pMatlab constructors are overloaded to take a `map` as an argument, and return a distributed array.

A =

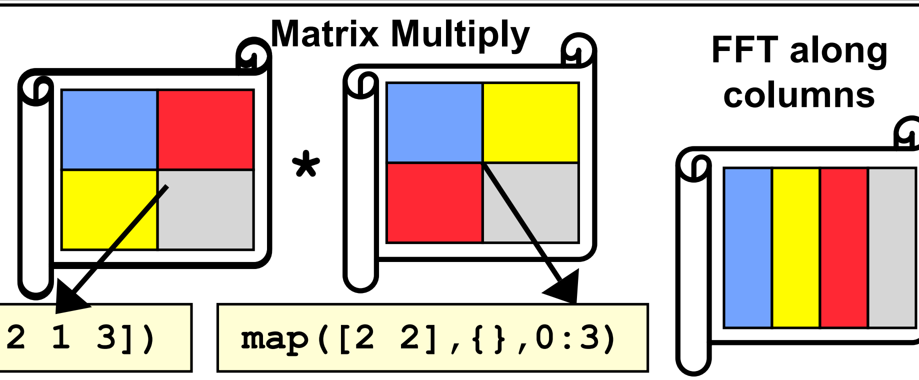
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Advantages of Maps

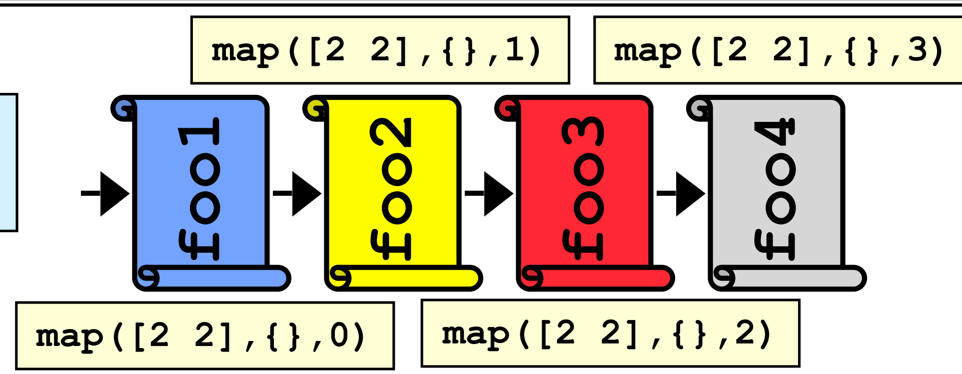
Maps are scalable. Changing the number of processors or distribution does not change the application.



Maps support different algorithms. **Different parallel algorithms have different optimal mappings.**



Maps allow users to set up pipelines in the code (implicit task parallelism).



Redistribution of Data

Math

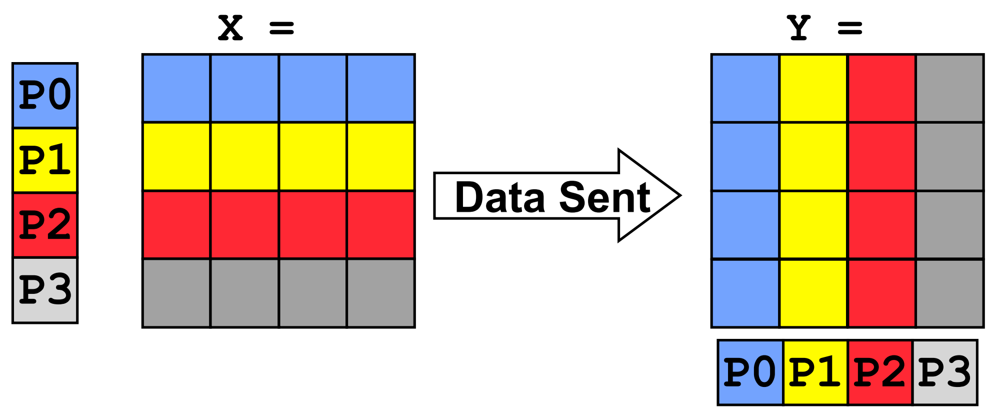
$$\mathbf{X} : \mathbb{R}^{P(N) \mathrm{x} N}$$
$$\mathbf{Y} : \mathbb{R}^{N \mathrm{x} P(N)}$$

$$\mathbf{Y} = \mathbf{X} + 1$$

pMatlab

```
Xmap = map([Np 1],{},0:Np-1);
Ymap = map([1 Np],{},0:Np-1);
X = zeros(N,N,Xmap);
Y = zeros(N,N,Ymap);

Y(:,:) = X + 1;
```



- Different distributed arrays can have different maps
- Assignment between arrays with the “=” operator causes data to be redistributed
- Underlying library determines all the message to send

Outline

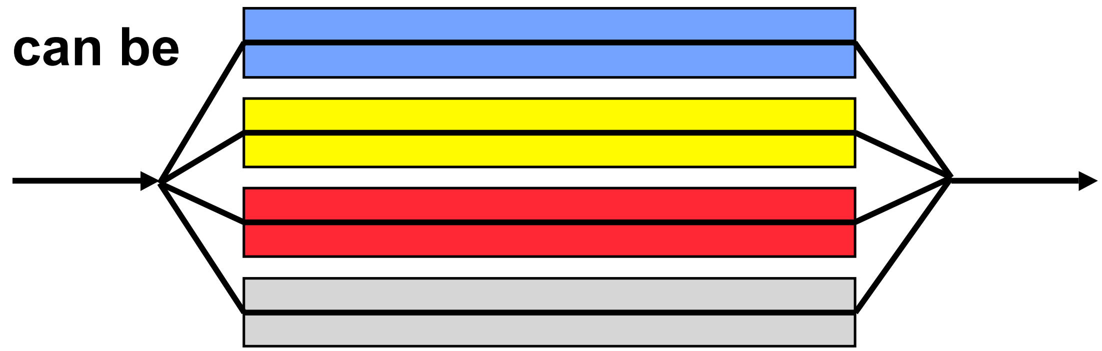
- Parallel Design
- Distributed Arrays
- **Concurrency vs Locality**
  - *Definition*
  - *Example*
  - *Metrics*
- Execution
- Summary

Definitions

<u>Parallel Concurrency</u>

- **Number of operations that can be done in parallel (i.e. no dependencies)**
- **Measured with:**

  Degrees of Parallelism

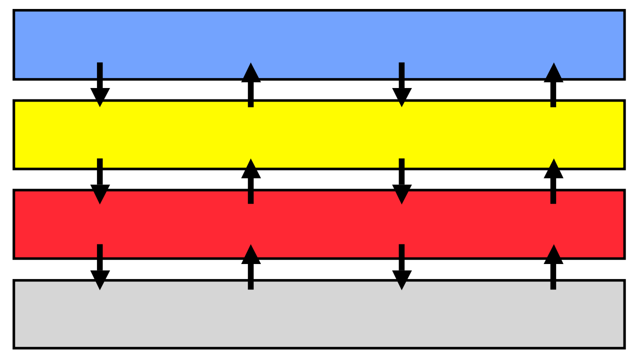


<u>Parallel Locality</u>

- **Is the data for the operations local to the processor**
- **Measured with ratio:**

  Computation/Communication

  = (Work)/(Data Moved)



- **Concurrency is ubiquitous; “easy” to find**
- **Locality is harder to find, but is the key to performance**
- ***Distributed arrays derive concurrency from locality***

Serial

Math	Matlab

Math

$$\mathbf{X},\mathbf{Y} : \mathbb{R}^{N x N}$$

$$\begin{array}{l} \textbf{for } i=1{:}N \\ \quad \textbf{for } j=1{:}N \\ \qquad \mathbf{Y}(i,j) = \mathbf{X}(i,j) + 1 \end{array}$$

Matlab

```
X = zeros(N,N);
Y = zeros(N,N);

for i=1:N
  for j=1:N
    Y(i,j) = X(i,j) + 1;
  end
end
```

- **Concurrency: max degrees of parallelism = N^2**
- **Locality**
  - Work = N^2
  - Data Moved: depends upon map

1D distribution

Math

$\mathbf{X},\mathbf{Y} : \mathbb{R}^{P(N)xN}$

for $i=1:N$
  for $j=1:N$
    $\mathbf{Y}(i,j) = \mathbf{X}(i,j) + 1$

pMatlab

```
XYmap = map([NP 1],{},0:Np-1);
X = zeros(N,N,XYmap);
Y = zeros(N,N,XYmap);

for i=1:N
  for j=1:N
    Y(i,j) = X(i,j) + 1;
  end
end
```

- **Concurrency: degrees of parallelism = min(N,N_P)**
- **Locality: Work = N^2, Data Moved = 0**
- **Computation/Communication = Work/(Data Moved) $\rightarrow \infty$**

2D distribution

Math	**pPMatlab**

Math

$\mathbf{X},\mathbf{Y} : \mathbb{R}^{P(N) \times P(N)}$

$$
\begin{aligned}
&\textbf{for } i=1:N \\
&\quad \textbf{for } j=1:N \\
&\qquad \mathbf{Y}(i,j) = \mathbf{X}(i,j) + 1
\end{aligned}
$$

pMatlab

```
XYmap = map([Np/2 2],{},0:Np-1);
X = zeros(N,N,XYmap);
Y = zeros(N,N,XYmap);

for i=1:N
  for j=1:N
    Y(i,j) = X(i,j) + 1;
  end
end
```

- **Concurrency: degrees of parallelism = min(N^2,N_P)**
- **Locality: Work = N^2, Data Moved = 0**
- **Computation/Communication = Work/(Data Moved) $\rightarrow \infty$**

2D Explicitly Local

Math

$\mathbf{X},\mathbf{Y} : \mathbb{R}^{P(N)xP(N)}$

for i=1:size(**X**.loc,1)
  for j=1:size(**X**.loc,2)
    Y.loc(i,j) =
      X.loc(i,j) + 1

pMatlab

```
XYmap = map([Np/2 2],{},0:Np-1);
Xloc = local(zeros(N,N,XYmap));
Yloc = local(zeros(N,N,XYmap));

for i=1:size(Xloc,1)
  for j=1:size(Xloc,2)
    Yloc(i,j) = Xloc(i,j) + 1;
  end
end
```

- **Concurrency: degrees of parallelism = min(N^2,N_P)**
- **Locality: Work = N^2, Data Moved = 0**
- **Computation/Communication = Work/(Data Moved) $\rightarrow \infty$**

1D with Redistribution

Math	pMatlab

Math

$\mathbf{X} : \mathbb{R}^{P(N)xN}$

$\mathbf{Y} : \mathbb{R}^{NxP(N)}$

for i=1:N
  for j=1:N
    $\mathbf{Y}(i,j) = \mathbf{X}(i,j) + 1$

pMatlab

```
Xmap = map([Np 1],{},0:Np-1);
Ymap = map([1 Np],{},0:Np-1);
X = zeros(N,N,Xmap);
Y = zeros(N,N,Ymap);

for i=1:N
  for j=1:N
    Y(i,j) = X(i,j) + 1;
  end
end
```

- **Concurrency: degrees of parallelism = min(N,N_P)**
- **Locality: Work = N^2, Data Moved = N^2**
- **Computation/Communication = Work/(Data Moved) = 1**

Outline

- Parallel Design
- Distributed Arrays
- Concurrency vs Locality
- **Execution**
  - *Four Step Process*
  - *Speedup*
  - *Amdahl's Law*
  - *Perforfmance vs Effort*
  - *Portability*
- Summary

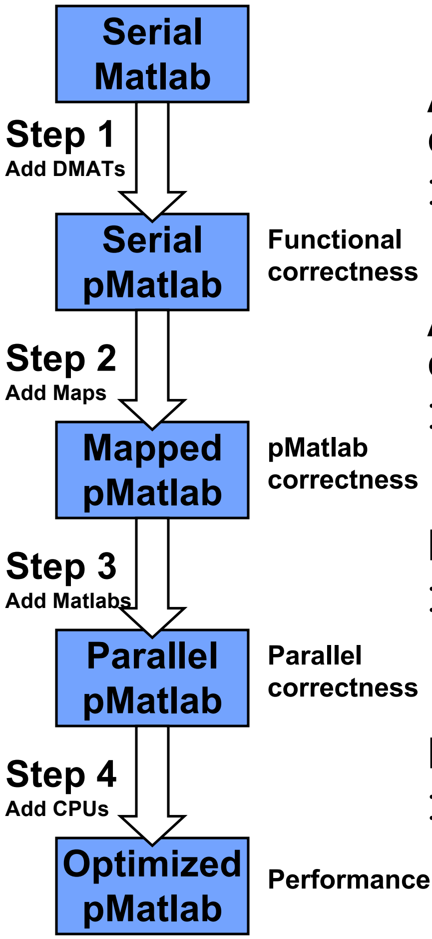
Running

- **Start Matlab**
  - Type: `cd examples/AddOne`
- **Run dAddOne**
  - Edit `pAddOne.m` and set: `PARALLEL = 0;`
  - Type: `pRUN('pAddOne',1,{})`
- **Repeat with:** `PARALLEL = 1;`
- **Repeat with:** `pRUN('pAddOne',2,{});`
- **Repeat with:** `pRUN('pAddOne',2,{'cluster'});`

- **Four steps to taking a serial Matlab program and making it a parallel Matlab program**

Parallel Debugging Processes

- Simple four step process for debugging a parallel program



Add distributed matrices without maps, verify functional correctness

```
PARALLEL=0;    pRUN('pAddOne',1,{});
```

Add maps, run on 1 processor, verify parallel correctness, compare performance with Step 1

```
PARALLEL=1;    pRUN('pAddOne',1,{});
```

Run with more processes, verify parallel correctness

```
PARALLEL=1;    pRUN('pAddOne',2,{}) );
```

Run with more processors, compare performance with Step 2

```
PARALLEL=1;    pRUN('pAddOne',2,{'cluster'});
```

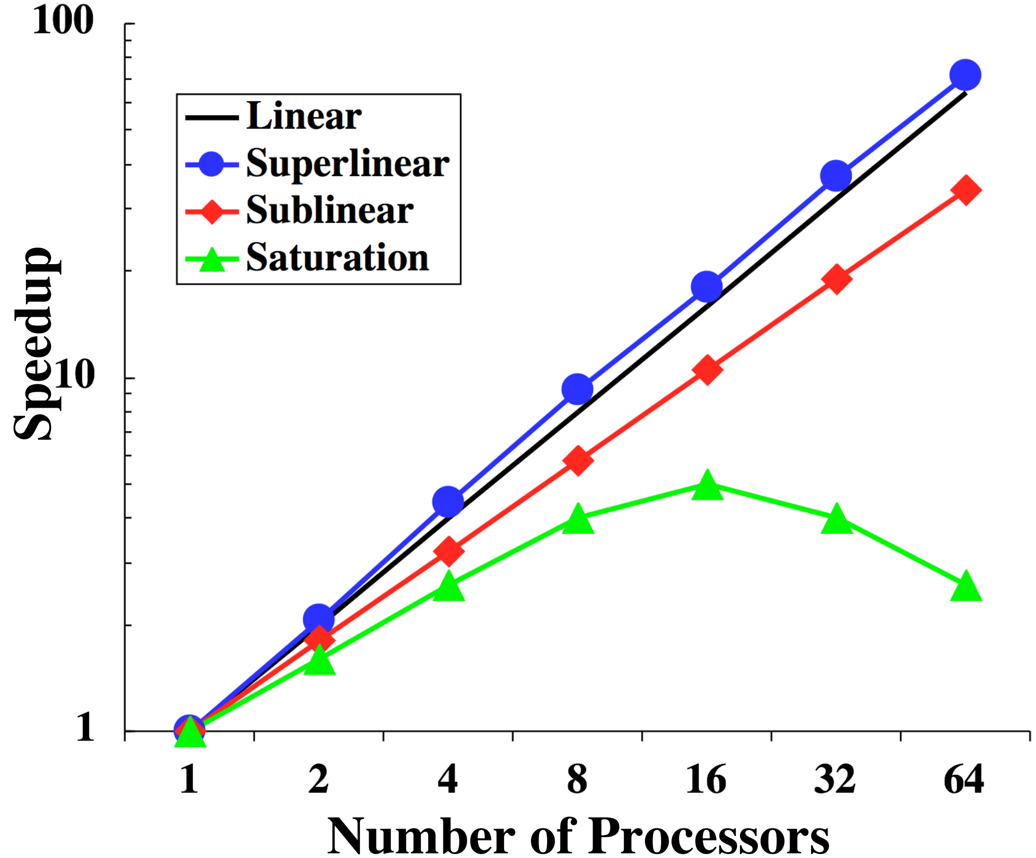
- **Always debug at earliest step possible (takes less time)**

Timing

- Run dAddOne: `pRUN('pAddOne',1,{'cluster'});`
  - Record `processing_time`
- Repeat with: `pRUN('pAddOne',2,{'cluster'});`
  - Record `processing_time`
- Repeat with: `pRUN('pAddone',4,{'cluster'});`
  - Record `processing_time`
- Repeat with: `pRUN('pAddone',8,{'cluster'});`
  - Record `processing_time`
- Repeat with: `pRUN('pAddone',16,{'cluster'});`
  - Record `processing_time`

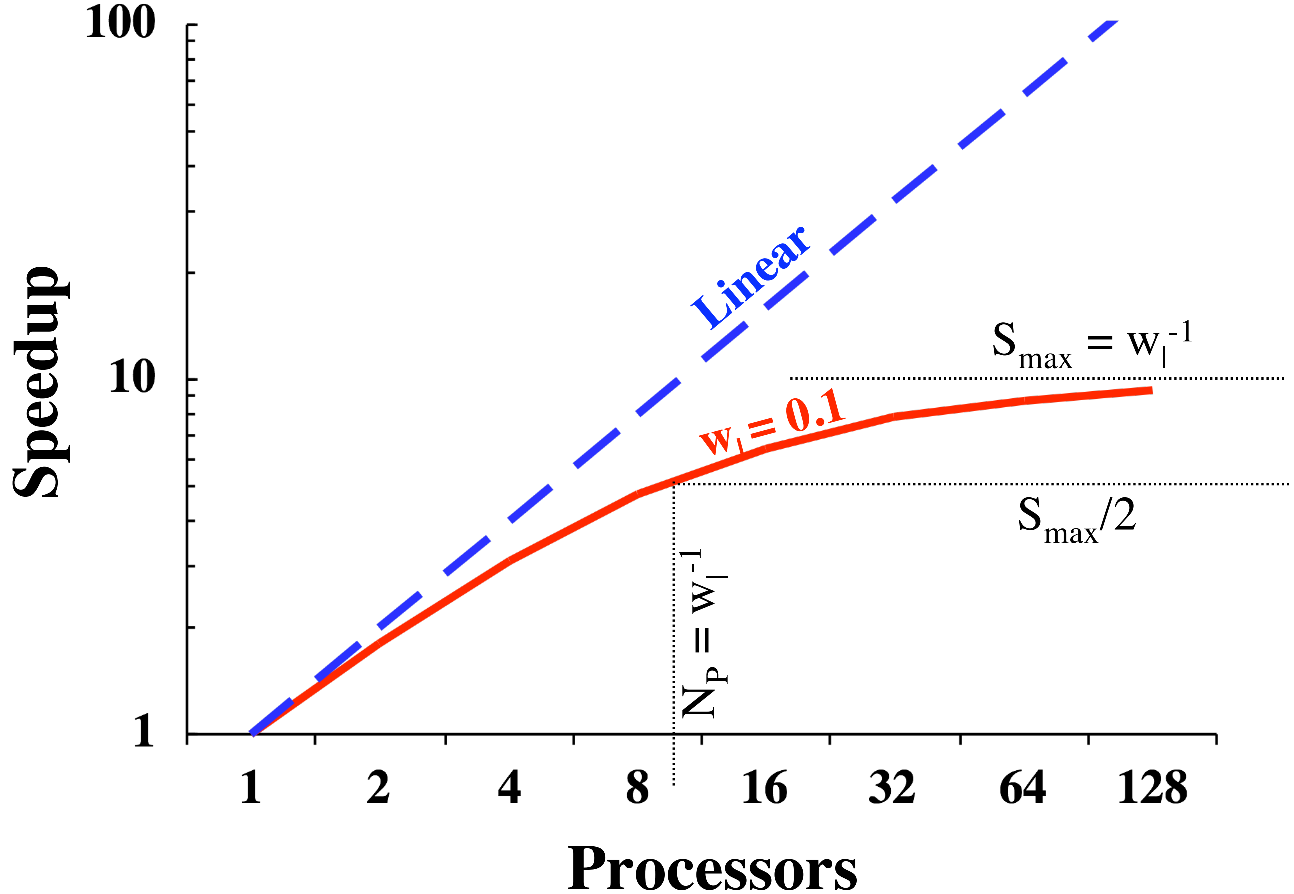
- **Run program while doubling number of processors**
- **Record execution time**

Computing Speedup



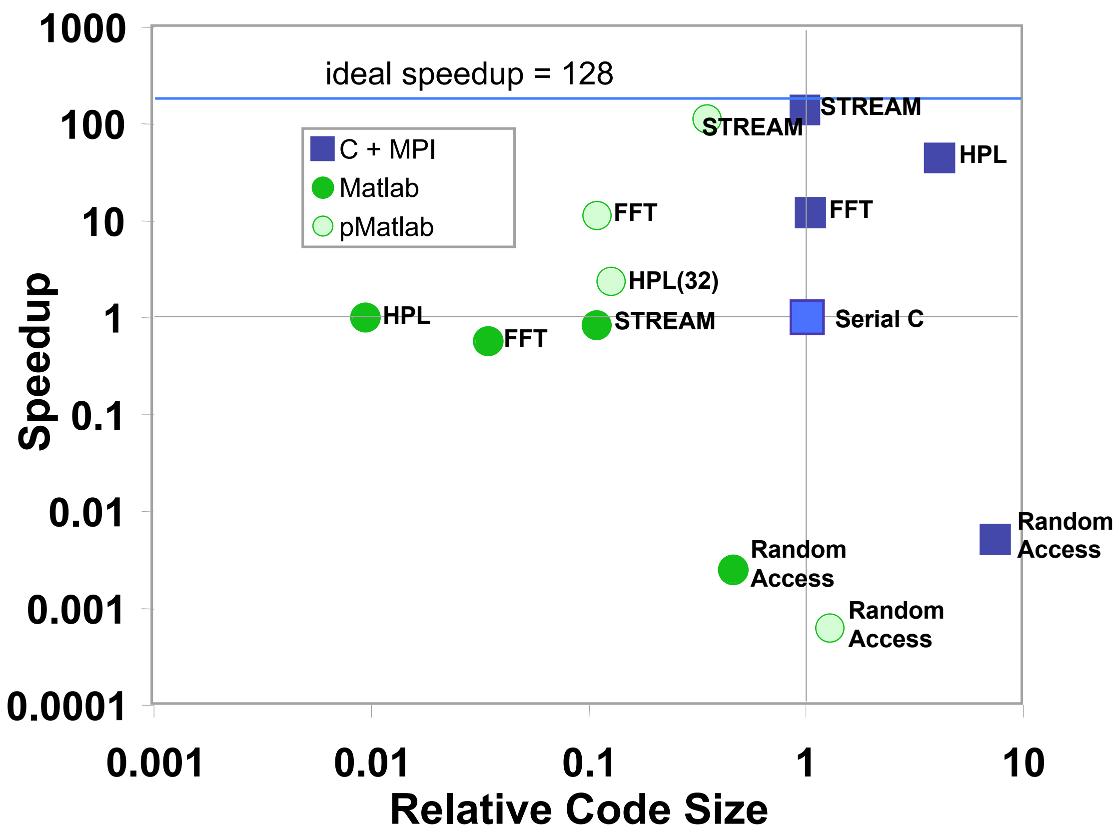
- Speedup Formula: Speedup(N_P) = Time(N_P=1)/Time(N_P)
- Goal is sublinear speedup
- All programs saturate at some value of N_P

Amdahl's Law



- Divide work into parallel ($w_{||}$) and serial ($w_|$) fractions
- Serial fraction sets maximum speedup: $S_{max} = w_|^{-1}$
- Likewise: Speedup(N_P=$w_|^{-1}$) = $S_{max}/2$

HPC Challenge Speedup vs Effort



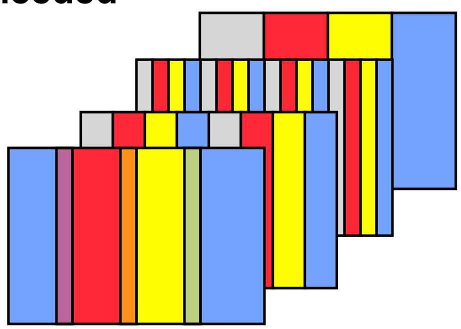
- **Ultimate Goal is speedup with minimum effort**
- **HPC Challenge benchmark data shows that pMatlab can deliver high performance with a low code size**

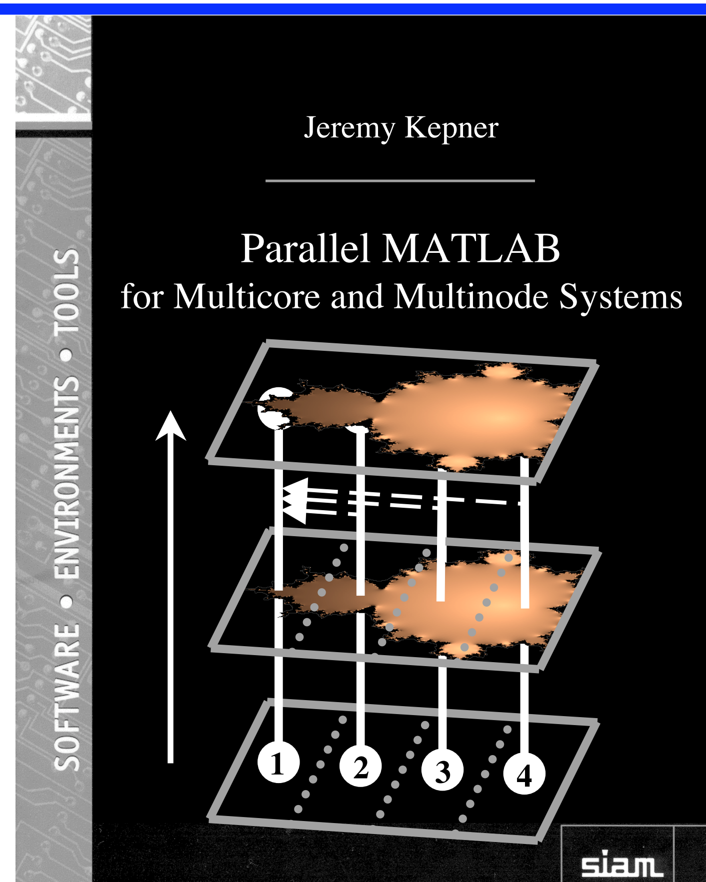
Portable Parallel Programming

Universal Parallel Matlab programming

```
Amap = map([Np 1],{},0:Np-1);
Bmap = map([1 Np],{},0:Np-1);
A = rand(M,N,Amap);
B = zeros(M,N,Bmap);
B(:,:) = fft(A);
```

- pMatlab runs in all parallel Matlab environments
- Only a few functions are needed
  - `Np`
  - `Pid`
  - `map`
  - `local`
  - `put_local`
  - `global_index`
  - `agg`
  - `SendMsg/RecvMsg`





- **Only a small number of distributed array functions are necessary to write nearly all parallel programs**
- **Restricting programs to a small set of functions allows parallel programs to run efficiently on the widest range of platforms**

Summary

- Distributed arrays eliminate most parallel coding burden
- Writing well performing programs requires expertise
- Experts rely on several key concepts
  - Concurrency vs Locality
  - Measuring Speedup
  - Amdahl’s Law
- Four step process for developing programs
  - Minimizes debugging time
  - Maximizes performance