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

![Graph showing performance speedup vs. programmer effort]
Outline

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

- Serial Program
- Parallel Execution
- Distributed Arrays
- Explicitly Local
parallel MATLAB

Serial Program

Math

\[ X, Y : \mathbb{R}^{N \times N} \]

\[ Y = X + 1 \]

Matlab

\[ X = \text{zeros}(N,N); \]
\[ Y = \text{zeros}(N,N); \]

\[ Y(:, :) = X + 1; \]

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

**Math**

\[ X, Y : \mathbb{R}^{\times N \times N} \]

\[ Y = X + 1 \]

**pMatlab**

\[ \text{Pid}=0 \]

\[ \text{Pid}=1 \]

\[ \text{Pid} = N_p - 1 \]

```matlab
X = zeros(N,N);
Y = zeros(N,N);
Y(:, :) = X + 1;
```

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

Math

\[ X, Y : \mathbb{R}^{P(N) \times N} \]

\[ Y = X + 1 \]

pMatlab

\[ \text{XYmap} = \text{map}([Np \ N1],\{\},0:Np-1); \]
\[ X = \text{zeros}(N,N,\text{XYmap}); \]
\[ Y = \text{zeros}(N,N,\text{XYmap}); \]
\[ Y(:,:,:) = X + 1; \]

- Use \( P() \) notation (or \text{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**

\[ X, Y : \mathbb{R}^{P(N) \times N} \]

\[ Y_{loc} = X_{loc} + 1 \]

**pMatlab**

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

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

- 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

Math

- $R^{P(N) \times N}$
- $R^{N \times P(N)}$
- $R^{P(N) \times P(N)}$

Matlab

- `Xmap=map([Np 1],{},0:Np-1)`
- `Xmap=map([1 Np],{},0:Np-1)`
- `Xmap=map([Np/2 2],{},0:Np-1)`
Maps and Distributed Arrays

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

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

A processor grid

\[ \text{A} = \text{zeros}(4, 6, \text{Amap}); \]

List of processors

Distribution

\{\} = \text{default} = \text{block}

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

```matlab
% Application
A = rand(M, map<i>);
B = fft(A);

map1 = map([Np 1], {}, 0:Np-1);
map2 = map([1 Np], {}, 0:Np-1);
```

```matlab
map([2 2], {}, [0 2 1 3])
map([2 2], {}, 0:3)
map([2 2], {}, 1)
map([2 2], {}, 3)
map([2 2], {}, 0)
map([2 2], {}, 2)
```
Redistribution of Data

Math

\[ X : \mathbb{R}^{P(N) \times N} \]

\[ Y : \mathbb{R}^{N \times P(N)} \]

\[ Y = X + 1 \]

pMatlab

\[
\begin{align*}
X_{\text{map}} &= \text{map}([Np \ 1],\{\},0:Np-1); \\
Y_{\text{map}} &= \text{map}([1 \ Np],\{\},0:Np-1); \\
X &= \text{zeros}(N,N,X_{\text{map}}); \\
Y &= \text{zeros}(N,N,Y_{\text{map}}); \\
Y(:, :) &= X + 1;
\end{align*}
\]

- 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

X:

P0
P1
P2
P3

Y:

P0
P1
P2
P3

Data Sent
Outline

- Parallel Design
- Distributed Arrays
- **Concurrency vs Locality**
  - Definition
  - Example
  - Metrics
- Execution
- Summary
Parallel Concurrency
• Number of operations that can be done in parallel (i.e. no dependencies)
• Measured with:
  Degrees of Parallelism

Parallel Locality
• 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
<table>
<thead>
<tr>
<th>Math</th>
<th>Matlab</th>
</tr>
</thead>
</table>
| \(X, Y : \mathbb{R}^{N \times N}\) | \(X = \text{zeros}(N,N);\)  
  \(Y = \text{zeros}(N,N);\) |
| \(\text{for } i=1:N\)  
  \(\text{for } j=1:N\)  
  \(Y(i,j) = X(i,j) + 1\) | \(\text{for } i=1:N\)  
  \(\text{for } j=1:N\)  
  \(Y(i,j) = X(i,j) + 1;\)  
  end |

- **Concurrency:** max degrees of parallelism = \(N^2\)
- **Locality**
  - Work = \(N^2\)
  - Data Moved: depends upon map
1D distribution

**Math**

\[ X, Y : \mathbb{R}^{p(N) \times N} \]

for \( i = 1:N \)
for \( j = 1:N \)
  \[ Y(i,j) = X(i,j) + 1 \]

**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

\[ X, Y : \mathbb{R}^{P(N) \times P(N)} \]

for i=1:N
  for j=1:N
    \[ Y(i,j) = X(i,j) + 1 \]
  end
end

\[ X, Y : \mathbb{R}^{P(N) \times P(N)} \]

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) → ∞
2D Explicitly Local

Math

\[ X, Y : \mathbb{R}^{P(N)} \times P(N) \]

\[
\begin{align*}
\text{for } i &= 1 : \text{size}(X.loc,1) \\
\text{for } j &= 1 : \text{size}(X.loc,2) \\
Y.loc(i,j) &= X.loc(i,j) + 1
\end{align*}
\]

\[
\text{for } i &= 1 : \text{size}(Xloc,1) \\
\text{for } j &= 1 : \text{size}(Xloc,2) \\
Yloc(i,j) &= Xloc(i,j) + 1
\]

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

\[
\text{for } i &= 1 : \text{size}(Xloc,1) \\
\text{for } j &= 1 : \text{size}(Xloc,2) \\
Yloc(i,j) &= Xloc(i,j) + 1;
\]

**Concurrency:** degrees of parallelism = min\( (N^2, N_p) \)

**Locality:** Work = \( N^2 \), Data Moved = 0

**Computation/Communication:** Computation/Communication = \( \text{Work}/(\text{Data Moved}) \) → ∞
1D with Redistribution

Math

\[ X : \mathbb{R}^{P(N) \times N} \]
\[ Y : \mathbb{R}^{N \times P(N)} \]

for i=1:N
  for j=1:N
    \[ Y(i,j) = X(i,j) + 1 \]
  end
end

pMatlab

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

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
  - Performance vs Effort
  - Portability
- Summary
Running

• Start Matlab
  – Type: \( \text{cd examples/AddOne} \)

• Run dAddOne
  – Edit \( \text{pAddOne.m} \) and set: \( \text{PARALLEL} = 0; \)
  – Type: \( \text{pRUN('pAddOne',1,{});} \)

• Repeat with: \( \text{PARALLEL} = 1; \)

• Repeat with: \( \text{pRUN('pAddOne',2,{});} \)

• Repeat with: \( \text{pRUN('pAddOne',2,{'cluster'});} \)

• 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

1. **Serial Matlab**
   - Add distributed matrices without maps, verify functional correctness
   - `PARALLEL=0; pRUN('pAddOne', 1, {})`

2. **Serial pMatlab**
   - Add maps, run on 1 processor, verify parallel correctness, compare performance with Step 1
   - `PARALLEL=1; pRUN('pAddOne', 1, {})`

3. **Mapped pMatlab**
   - Run with more processes, verify parallel correctness
   - `PARALLEL=1; pRUN('pAddOne', 2, {})`

4. **Parallel pMatlab**
   - 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: \( \text{Speedup}(N_P) = \frac{\text{Time}(N_P=1)}{\text{Time}(N_P)} \)
- Goal is sublinear speedup
- All programs saturate at some value of \( N_P \)
Amdahl’s Law

- Divide work into parallel ($w_\parallel$) and serial ($w_i$) fractions
- Serial fraction sets maximum speedup: $S_{\text{max}} = w_i^{-1}$
- Likewise: $\text{Speedup}(N_P=w_i^{-1}) = S_{\text{max}}/2$
**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**
Universal Parallel Matlab programming

\[
\begin{align*}
\text{Amap} & = \text{map}([\text{Np} \ 1], \{\}, 0:\text{Np}-1); \\
\text{Bmap} & = \text{map}([1 \ \text{Np}], \{\}, 0:\text{Np}-1); \\
\text{A} & = \text{rand}(\text{M}, \text{N}, \text{Amap}); \\
\text{B} & = \text{zeros}(\text{M}, \text{N}, \text{Bmap}); \\
\text{B}(:,:,:) & = \text{fft}(\text{A});
\end{align*}
\]

- 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

Get It Right

- Step 1: Serial MATLAB
  - Functional correctness
  - Add DMATs

- Step 2: Serial pMatlab
  - pMatlab correctness
  - Add Maps

- Step 3: Mapped pMatlab
  - Parallel correctness
  - Add Matlabs

- Step 4: Parallel pMatlab
  - Performance
  - Add CPUs

Make It Fast