The **student-driven HPC** environment at **SLIM**

Tim Lin, Brazil IIP FWI Workshop 2015
Conservation of complexity...

... in terms of a student’s focus, attention, time
theoretically inclined student
computationally inclined student
area = time spent in program
Trilemma of techniques explored

Choose 2
Trilemma of techniques explored

Choose 2
Trilemma of techniques explored

Choose 2

HARD TO MAINTAIN

sophisticated

PURE MATLAB

easy
to implement

runs fast

PURE MATLAB
Trilemma of techniques explored

Choose 2

- ALREADY DONE
- PURE MATLAB
- UNMAINTAINABLE CRAZINESS

- sophisticated
- runs fast
- easy to implement

ALREADY DONE

Papers
Pairing a “theoretical” and a “technical” person

Low-level fork
Projects get taken over and eventually stagnate

tweaks

different tweaks
Projects get taken over and eventually stagnate
Projects get taken over and eventually stagnate

low-level fork
unmaintainable
The “dream”
But which language/package?

Why use the same language or packages?
**algorithm design**

flexible  
easy to read  
easy to debug  
reflects math  
encourage experiments

**computation engine**

well-defined behaviour  
mature compiler  
low-level access  
allows tweaking  
parallel systems
algorithm design

computation engine

Call

Signal
algorithm design

shell scripts

Call

seismic utilities

Signal

computation engine
algorithm design

C/C++/F90

Call

Signal

computation engine

BLAS/FFTW/MKL
algorithm
design

MATLAB

Call

Signal

LAPACK/ScaLAPACK/
MEX_files

computation
game
algorithm
design

MATLAB

computation
ingine

Call

LAPACK/ScaLAPACK/
MEX_files

Signal

Abstraction layer is important
Currently, we program in MATLAB, but our abstraction for distributed computation is based on Parallel Matlab (PCT) and pSPOT.
“Parallel Matlab”

Officially another “toolbox” on top of Matlab, called “Parallel Computing Toolbox” or PCT

Two components:

• The “toolbox” itself, which provides the parallelization code and can spawn local workers

• The “Distributed Compute Server” (MDCS) which allows spawning workers on external nodes in a cluster
  • can bring own scheduler, i.e., SLIM uses Torque, SENAI uses Slurm
The search for parallel arrays

Previous to 2009, SLIM was mostly based on SciPy/NumPy computing kernel with a traditional command-line seismic processing interface (at the time, RSF/Madagascar)

Developed a symbolic math DSL (SLIMpy) which translates mathematical expressions to an AST that writes out shell scripts that call RSF programs
  • also write out reproducible SCons scripts (Make-equivalent)

But.... hard to parallelize non-trivially
The search for parallel arrays

_Wanted_: A true framework for shared-memory-like distributed arrays, which works similarly to NumPy arrays and is interactive

Two candidates
- Star-P (for Python)
- Matlab PCT (at the time just added distributed arrays)

Went with Matlab PCT on a hunch (actually it was cheaper)
- eventually Star-P acquired by Microsoft and “sunsetted”
Matlab PCT operation

Always assumes a “master” supervisor for a pool of workers

Each worker (and master) are independent, complete Matlab processes, and communicate via a MPI-based backend

Workers form a “pool” that can be provisioned and released interactively from the command line

Local workers free, individual licensing price for remote workers
Distributed arrays

Emulates a normal numeric array
- by default distributed evenly across the last dimension
- APIs to change underlying distribution
- can be constructed in many ways... from simple to complex
- easy way to learn about shared-memory/NUMA type architecture

Killer Feature: overloading of many Matlab functions on local numeric arrays to distributed arrays
| atan          | csc          | horzcat([]) | min         | realpow      | triu         |
| atanh        | cscd         | hsv2rgb     | minus(-)    | realsqrt     | true         |
| atand        | csch         | hypot        | mldivide(\) | rem          | typecast     |
| atan2        | ctranspose(') | ifft         | mrdivide(⁄) | repmat       | uint16       |
| besselh      | cummax       | ifft2        | mtimes(*)   | reshape      | uint32       |
| besseli      | cummin       | ifftn        | mod         | rgb2hsv      | uint64       |
| besselj      | cumprod      | imag         | mode        | rmfield      | uint8        |
| besselk      | cumsum       | Inf          | NaN         | round        | uminus(-)    |
| bessely      | diag         | int16        | ndims       | sec          | unwrap       |
| beta         | diff         | int32        | ndgrid      | secd         | uplus(+)     |
| betainc      | dot          | int64        | ne(\(\cong\)) | sech        | vander       |
| betaincinv   | double       | int8         | nextpow2    | sign         | var          |
| betaln       | eig          | inv          | nnz         | sin          | vertcat([])  |
| bitand       | end          | ipermute     | nonzeros    | sind         | xor          |
| bitor        | eps          | isempty      | norm        | single       | zeros        |
| bitxor       | eq(\(\cong\)) | isequal      | normest     | sinh         |              |
| bsexfun      | erf          | isequaln     | not(\(~\)) | size         |              |
| cart2pol     | erfc         | isfinite     | nargout     | sort         |              |
|              | erfcinv      | isfloat      | num2cell    | sortrows     |              |
Distributed arrays

Allows many existing codebase to work directly on distributed arrays with very few changes

- improved collaboration with outsiders
- brute effort provided by Mathworks, continuous improvement
- trading licensing fee for student time
- vastly improved maintainability from being able to limit code branching for parallel mode
“Not that slow”

HPCC High-Performance Linpack benchmark in three lines:

```matlab
A = distributed.randn(m, m, distributor2dbc);
b = distributed.rand(m, 1);
tic
x = A\b;
toc;
```

Just ran small benchmark on YEMOJA:
- 128 nodes, 8 process (1024 total)
- inverting 1,800,000-by-1,800,000 matrix (380 GB)
- took 623 seconds, about 12 TFlop/s
“Not that slow”

Verification code as per HPCC spec

```matlab
% Compute scaled residuals
r1 = norm(A*x-b,inf)/(eps*norm(A,1)*m);
r2 = norm(A*x-b,inf)/(eps*norm(A,1)*norm(x,1));
r3 = norm(A*x-b,inf)/(eps*norm(A,inf)*norm(x,inf)*m);

if max([r1 r2 r3]) > 16
    error('Failed the HPC HPL Benchmark');
end
```
SPOT/pSPOT built on distributed array

A way to encapsulate kernel computations of linear operations into something that “looks like a matrix”

\[ F = \text{opDFT}(512) \]
\[ x = \text{randn}(512,1) \]
\[ xf = F \ast x \]
\[ x == F' \ast xf \]

Inherently express the notion of multilinear transformations on tensors into Kronecker products

\[ FK = \text{opKron} \left( \text{opDFT}(300), \text{opDFT}(512) \right) \]
\[ x = \text{randn}(512,300) \]
\[ x_{\text{fk}} = FK \ast x( :) \]
\[ x == F' \ast xf \]
SPOT/pSPOT built on distributed array

Extends to distributed paradigm (implicitly performs transpose)

```plaintext
F = opDFT(1024);
F2D = opKron(F,F);
F4D = oppKron2Lo(F2D,F2D);
F5D = oppKron2Lo(F2D,opKron(F,F,F));
x = distributed.randn(1024*1024*1024,1024*1024);
xf = F5D * x(:);
```
Non-separable example

Frequency-dependent filtering

\[
A = \text{oppDistFun}(f, \text{@filter})
\]

- \(f\) is (distributed) array of frequencies
- \(@\text{filter}(x,f)\) performs filter on \(x\) based on frequency \(f\)

Slice-wise matrix-matrix multiply

\[
A = \text{oppDistFun}(\text{MAT}, \text{@matmult})
\]

- \(\text{MAT}\) is 3D array distributed over the “slice” dim
- \(@\text{matmult}(x,\text{mat})\) performs mat-mult between \(x\) and \(\text{mat}\)
Non-separable example

Implemented SRME multiple prediction step using this framework

Just did small benchmark on YEMOJA again
  • 128 nodes, 8 process (1024 total)
  • Seismic line data: 2200 time samples, 2200 shots, 2200 trace/shot
  • ~42GB of data in single precision
  • SRME prediction finished in 751 seconds (~70 GFlop/s) using FFT
Parfor

A simple way to do parallel loops

Concept of a parallel index variable, Matlab AST parser will enforce that you do not use it to index
  • except for associative reduction operations and functions
  • a simple way to learn about map reduce for students
  • latest Matlab can also connect to Hadoop for “real” mapreduce
**Single program, multiple data** paradigm

- Each worker has local execution space using variable of same name
- Master has access to all worker’s local results outside of SPMD context, workers can also communicate
- Very easy to establish barriers and broadcasts
- Easy way for students to work in **UPC/BSP** paradigm, with **MPI** primitive equivalents available
- works well with distributed arrays (can access local part)
available from the parallel pool. If there are not enough workers available, an error is thrown. If n is zero, MATLAB executes the block and creates Composite objects, the same as if there is no pool available.

spmd(m,n), statements, end uses a minimum of m and a maximum of n workers to evaluate statements. If there are not enough workers available, an error is thrown. m can be zero, which allows the block to run locally if no workers are available.

For more information about spmd and Composite objects, see Distributed Arrays and SPMD.

Examples

Perform a simple calculation in parallel, and plot the results:

```
parpool(3)
spmd
    % build magic squares in parallel
    q = magic(labindex + 2);
end
for ii=1:length(q)
    % plot each magic square
    figure, imagesc(q{ii});
end
delete(gca)
```

More About

- An spmd block runs on the workers of the existing parallel pool. If no pool exists, spmd will start a new parallel pool, unless the starting of pools is disabled in your parallel preferences. If there is no parallel pool and spmd cannot start one, the code runs in serial.
Detailed communication control btw workers

Many message-passing communication routines from MPI are exposed in a Matlab way
# Task Control and Worker Communication

Control task code execution and communication among workers during job and spmd block execution

## Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>labindex</td>
<td>Index of this worker</td>
</tr>
<tr>
<td>numlabs</td>
<td>Total number of workers operating in parallel on current job</td>
</tr>
<tr>
<td>gcat</td>
<td>Global concatenation</td>
</tr>
<tr>
<td>gop</td>
<td>Global operation across all workers</td>
</tr>
<tr>
<td>gplus</td>
<td>Global addition</td>
</tr>
<tr>
<td>pload</td>
<td>Load file into parallel session</td>
</tr>
<tr>
<td>psave</td>
<td>Save data from communicating job session</td>
</tr>
<tr>
<td>labBarrier</td>
<td>Block execution until all workers reach this call</td>
</tr>
<tr>
<td>labBroadcast</td>
<td>Send data to all workers or receive data sent to all workers</td>
</tr>
<tr>
<td>labProbe</td>
<td>Test to see if messages are ready to be received from other worker</td>
</tr>
<tr>
<td>labReceive</td>
<td>Receive data from another worker</td>
</tr>
<tr>
<td>labSend</td>
<td>Send data to another worker</td>
</tr>
<tr>
<td>labSendReceive</td>
<td>Simultaneously send data to and receive data from another worker</td>
</tr>
<tr>
<td>getCurrentJob</td>
<td>Job object whose task is currently being evaluated</td>
</tr>
<tr>
<td>getCurrentCluster</td>
<td>Cluster object that submitted current task</td>
</tr>
<tr>
<td>getCurrentTask</td>
<td>Task object currently being evaluated in this worker session</td>
</tr>
</tbody>
</table>
\( FUN(FUN(x1,x2),x3) = FUN(x1,FUN(x2,x3)) \)

\( \text{res} = \text{gop}(\text{FUN},x,\text{targetlab}) \) performs the reduction, and places the result into \( \text{res} \) only on the worker in

**Examples**

This example shows how to calculate the sum and maximum values for \( x \) among all workers.

```matlab
p = parpool('local',4);
x = Composite();
x{1} = 3;
x{2} = 1;
x{3} = 4;
x{4} = 2;
spmd
  xsum = gop(@plus,x);
xmax = gop(@max,x);
end
xsum{1}
```

10

```matlab
xmax{1}
```

4

This example shows how to horizontally concatenate the column vectors of \( x \) from all workers into a matrix. It w
Conclusions

- Student time remains constant over the years, but things to learn increase faster and faster each year
- Abstractions save us from inevitable specialization
- Using high-level abstraction and easy tools for parallel computation, students can save programming time and use it on other topics that are also becoming increasingly complex
- Encourages collaboration by minimizing mundane parts of the codebase, many cases serial and parallel program can share the same codes
- *Not* for free: enforces good programming style and separation of concern for the code
- This paradigm great increased scientific productivity at SLIM