Parallel Computing with MATLAB

Jos Martin  Principal Architect, Parallel Computing Tools  jos.martin@mathworks.co.uk
Code used in this presentation can be found at

Overview

- Scene setting
- Task Parallel (*par*)
- Why doesn’t it speed up as much as I expected?
- Data parallel (*spmd*)
- GPUs
What I assume

- Reasonable MATLAB knowledge
  - e.g. vectorization, pre-allocation
- Some use of PCT and associated concepts
  - What is a cluster
  - Simple `parfor` usage
parfor

Definition

*Code in a **parfor** loop is guaranteed by the programmer to be execution order independent*

Why is that important?

We can execute the iterates of the loop in any order, potentially at the same time on many different workers.
A simple `parfor` loop

```matlab
parfor i = 1:N
    out(i) = someFunction(in(i));
end
```
parfor – how it works

- A loop from 1:N has N *iterates* which we partition into a number of *intervals*
  - Each *interval* may have a different number of *iterates*

- Allocate the *intervals* to execute on the workers

- Stitch the results back together
The Mechanics of `parfor` Loops

```matlab
a = zeros(10, 1);
parfor i = 1:10
    a(i) = i;
end
```

Pool of MATLAB Workers
reduce = 0; bcast = ...; in = ...;
parfor i = 1:N
    temp = foo1(bcast, i);
    out(i) = foo2(in(i), temp);
    reduce = reduce + foo3(temp);
end
reduce = 0; bcast = ...; in = ...;
parfor i = 1:N
    temp = foo1(bcast, i);
    out(i) = foo2(in(i), temp);
    reduce = reduce + foo3(temp);
end
Making extra parallelism

- No one loop appears to have enough iterations to go parallel effectively

```matlab
for ii = 1:smallNumber_I
    for jj = 1:smallNumber_J
        for kk = 1:smallNumber_K
            end
        end
    end
end

smallNumber_I * smallNumber_J * smallNumber_K == quiteBigNumber
```
reduce = 0; bcast = ...; in = ...;
parfor i = 1:N
    temp = foo1(bcast, i);
    out(i) = foo2(in(i), temp);
    reduce = reduce + foo3(temp);
end
Broadcast variable

reduce = 0; \texttt{bcast} = \ldots; \texttt{in} = \ldots;
\parfor i = 1:N
    temp = \texttt{foo1(bcast, i)};
    out(i) = \texttt{foo2(in(i), temp)};
    reduce = reduce + \texttt{foo3(temp)};
\end
Reusing data

\[ D = \text{makeSomeBigData}; \]

\[ \text{for } ii = 1:N \]
\[ \hspace{1em} \text{parfor } jj = 1:M \]
\[ \hspace{2em} a(jj) = \text{func}(D, jj); \]
\[ \text{end} \]
\[ \text{end} \]
Reusing data

D = WorkerObjectWrapper(@makeSomeBigData);
for ii = 1:N
    parfor jj = 1:M
        a(jj) = func(D.value, jj);
    end
end

from Edric Ellis on MATLAB Central
www.mathworks.com/matlabcentral/fileexchange/31972-worker-object-wrapper
Counting events in parallel

- Inside the parallel loop you are looking to count the number of times some particular result is obtained
  - Histograms, interesting results, etc.
Reduction Variable

\[ \text{reduce} = 0; \quad \text{bcast} = \ldots; \quad \text{in} = \ldots; \]

\text{parfor} \ i = 1:N

\[
\begin{align*}
\text{temp} &= \text{foo1(bcast, i)}; \\
\text{out}(i) &= \text{foo2(in}(i), \text{ temp}); \\
\text{reduce} &= \text{reduce} + \text{foo3(temp)};
\end{align*}
\]

\text{end}
Common parallel program

set stuff going
while not all finished {
    for next available result do something;
}

Task Parallel (parfeval)
parfeval

- New feature in R2013b
- Introduces asynchronous programming

\[ f = \text{parfeval}(\text{@func}, \text{numOut}, \text{in1}, \text{in2}, \ldots) \]

- The return \( f \) is a future which allows you to
  - Wait for the completion of calling \( \text{func}(\text{in1}, \text{in2}, \ldots) \)
  - Get the result of that call
  - \( \ldots \) do other useful parallel programming tasks \( \ldots \)
Fetch Next

- Fetch next available unread result from an array of futures.

\[ [idx, out1, ...] = \text{fetchNext}(\text{arrayOfFutures}) \]

- `idx` is the index of the future from which the result is fetched
- Once a particular future has returned a result via `fetchNext` it will never do so again
  - That particular result is considered read, and will not be re-read

Task Parallel (`parfeval`)
Common parallel program (MATLAB)

% Set stuff going
for ii = N:-1:1
    fs(ii) = parfeval(@stuff, 1);
end
% While not all finished
for ii = 1:N
    % for next available result
    [whichOne, result] = fetchNext(fs);
    doSomething(whichOne, result);
end

parfevalWaitbarDemo
Better parallel program

```matlab
set N things going
while not all finished {
    set N more things going
    for N {
        for next available result do something;
    }
}
parfevalNeedleDemo
```
Why isn’t it as fast as I expect?

- How fast did you expect?
  - Why?

- Consider
  - Data transfer
  - Resource contention
  - Other overheads
Data Transfer

- **parfor** (Variable classification)
  - Broadcast goes once to each worker (what is actually accessed?)
  - Sliced sends just the slice (is all of the slice accessed?)
  - Reduction is sent back once per worker (usually efficient)

- **parfeval**
  - All inputs for a given call are passed to that worker
Resource Contention

![Diagram showing resource contention in a computer system]

- Memory
- Cache Memory (L3)
- Core
- HT
- IO Hub
- Disk
- Network
- Performance

26
Speedup vs. num. Concurrent Processes

\[ a = \text{bigMatrix} \]

\[ a*a \]

\[ \text{fft}(a) \]

\[ \text{sum}(a) \]
Speedup vs. num. Concurrent Processes

\[ a = \text{bigMatrix} \]

\[ a*a \]

\[ \text{fft}(a) \]

\[ \text{sum}(a) \]
Speedup vs. Size of Data (6 procs.)

\[ a = \text{matrix}(N) \]

\[ a \times a \]

\[ \text{sum}(a) \]

\[ \text{svd}(a) \]
Summary (par*)

- Find enough parallelism
  - Go parallel as soon as possible
  - But not too small with parfeval

- Know how much data is being sent
  - Try to send as little as possible

- Understand how multiple algorithms might interact

- Keep workers busy if possible
Single Program, Multiple Data (\texttt{spmd})

- Everyone executes the same program
  - Just with different data
  - Inter-lab communication library enabled
  - \texttt{labindex} and \texttt{numlabs} available to distinguish labs

- Example

\begin{verbatim}
x = 1
spmd
    y = x + labindex;
end
\end{verbatim}
A Mental Model for `spmd` ... `end`

Data Parallel (`spmd`)
Common Parallel Program

```plaintext
forever {
    results = independentStuff( params )
    if results are OK {
        break
    } else {
        params = chooseNewParams( results, params )
    }
}
```
Solve with `parfor`

```matlab
forever {
    parfor ii = 1:N {
        results(ii) = independentStuff(params(ii))
    }
    if results are OK {
        break
    } else {
        params = chooseNewParams(results, params)
    }
}
```
Solve with `spmd`

```matlab
spmd { forever {
    // Each of the workers computes its results (mine)
    results = gcat(independentStuff( params(mine) ))
    if results are OK {
        break
    } else {
        params = chooseNewParams( results, params )
    }
}
```

`spmdDemo`
Summary (spmd)

- Required if inter-worker communication is needed for the algorithm
- Can provide better performance for some algorithms
GPUs

- Highly threaded
  - $10^6$ threads not uncommon

- Very fast memory access
  - 200GB/s (~8x best CPU)

- Peak performance (double)
  - 1TFlop (~3x best CPU)
Getting data to the GPU

- To make an array exist on the GPU

```matlab
g = gpuArray( dataOnCpu );
g = zeros( argsToZeros, 'gpuArray' );
g = ones( argsToZeros, 'uint8', 'gpuArray' );
```

- Supported types
  - All built-in numeric types
    ```
    [complex][[uint|int][8|16|32|64]|double|single]
    ```
Using `gpuArray`

- Honestly – it’s just like an ordinary MATLAB array
- Except that the methods that are implemented for it will run on the GPU (over 200 currently and growing)
  - Maybe some of these will be faster on your GPU

- Want to get the data back to the CPU
  
  ```
  c = gather(g);
  ```
GPUness spreads

function  [a, b, c] = example(d, e, f)

a = sin(d) + e;
b = cos(d) + f;
c = a + b + e + f;
function [a, b, c] = example(d, e, f)
% Imagine if the input d were on the GPU
a = sin(d) + e;
b = cos(d) + f;
c = a + b + e + f;
Getting data in the right place (new in 13b)

sIn = size(in);
out = in * eye(sIn) + ones(sIn);

- The problem is that \texttt{eye} and \texttt{ones} make data in CPU memory
  - And so we need to transfer data to the GPU (which is \textit{relatively slow})

\[
\text{out} = \text{in} \times \text{eye}(\text{sIn}, 'like', \text{in}) + \text{ones}(\text{sIn}, 'like', \text{in});
\]

- '\textit{like}' says make the data in the same place and as the same type as the prototype provided
Semantic work pattern: \texttt{gpuArray}

\[ D = A \times B + C \]
Lazy Evaluation

- Where possible we queue things up on the GPU and return back to the program immediately
  - We also try to amalgamate sets of operations together
Actual work pattern: \texttt{gpuArray}

On GPU

- $A$
- $B$
- $C$
- $D_{\text{future}}$
- $D_{\text{actual}}$

On CPU

- $\text{tmp}_{\text{future}}$
- $\text{tmp}_{\text{actual}}$

CPU code continues

Time
Lazy Evaluation

- Why do you care?
  - Improves performance a lot
  - CPU & GPU work at the same time.
- But be careful because `tic;toc;` can easily give you the wrong time, since the computation hasn’t finished

```matlab
d = gpuDevice; % Get the current GPU device
tic
gpuStuffToTime;
wait(d); % wait for computation on the GPU d to finished
toc
```
Can we do better?

\[ D = A \cdot B + C \]
arrayfun

- Apply a function to each element of a set of gpuArrays

  \[ o1, o2 ] = \text{arrayfun}(\text{@aFunction}, s1, s2, s3) \]

- Some limitations apply
  - All code uses scalar variables
  - Only a subset of the MATLAB language is supported
Why is this a good idea?

- We know what inputs are being passed to your function
- We know what code is in your function

  with that we can infer the type of all variables in your code

- and then we can generate code for your GPU

- for each element of your input arrays we can execute your function on a single CUDA thread
  - remember a GPU can execute thousands of threads at once, and schedule even more
Singleton Expansion

Whenever a dimension of an input array is singleton (equal to one), we virtually replicates that array along that dimension to match the other arrays.

– *scalar expansion* is a specific instance of *singleton expansion*

Look for functions that support singleton expansion (*arrayfun*, etc.)
Batching many small operations (pagefun)

- You have many matrices held in the pages of a multi-dimensional array
- You want to carry-out the same operation on each of the individual pages of the big array e.g.

```matlab
for ii = 1:numPages
    C(:,:,ii) = A(:,:,ii) * B;
end
```
Invoking CUDA Kernels

MATLAB

% Setup
kern = parallel.gpu.CUDAKernel('myKern.ptx', cFcnSig)

% Configure
kern.ThreadBlockSize=[512 1];
kern.GridSize=[1024 1024];

% Run
[c, d] = feval(kern, a, b);

C & mex

// Setup
mxGPUArray const * A = mxGPUCreateFromMxArray(prhs[0]);
// Create a GPUArray to hold the result and get its underlying
// pointer.
mxGPUArray * B = mxGPUCreateGPUArray(mxGPUGetNumberOfDimensions(A),
    mxGPUGetDimensions(A),
    mxGPUGetClassID(A),
    mxGPUGetComplexity(A),
    MX_GPU_DO_NOT_INITIALIZE);
double * d_B = (double *)(mxGPUGetData(B));
// Standard CUDA kernel call using the CUDA runtime.
TimesTwo<<<blocksPerGrid, threadsPerBlock>>>(d_B, N);

// Device code prototype ...
void __global__ TimesTwo(double * const B, int const N) { ... };
Summary (GPU)

- Vectorize as much as possible
- Performance better for larger arrays (overhead smaller)
- Keep data on the GPU as long as possible
- Look for opportunities to use `arrayfun` and `pagefun`
  - Particularly some loops can become serial calls to these functions
  - Use less memory with singleton expansion