HPEC Challenge SAR Benchmark: 
pMatlab Implementation and Performance

Julia S. Mullen  
Theresa Meuse  
Jeremy Kepner  

September 20, 2006

This work is sponsored by the Department of the Air Force 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
Outline

• Introduction
• Parallel Strategies
  – Coarse
  – Fine grained
  – Pipelined
• Results
• Summary
HPEC Challenge: SAR Benchmark

- Key part of HPEC Challenge
- End-to-End Benchmark
- Parallel extension of SAR Benchmark Specification needed
- Prototype and test parallel strategies using pMatlab

SAR Benchmark

The HPGS Scalable Synthetic Compact Application #2 (SSCA #2) simulates a sensor processing chain (Figure 1). It consists of a front-end sensor processing stage, where Synthetic Aperture Radar (SAR) images are formed, and a back-end knowledge formation stage, where detection is performed on the difference of the SAR images. It generates its own synthetic 'raw' data, which is scalable. The goal is to mimic the most taxing computation and I/O requirements found in many embedded systems, such as medical/space imaging, or reconnaissance monitoring. Its principal performance goal is throughput, in other words, to maximize the rate at which answers are generated. The computational kernels must keep up with copious quantities of sensor data. Its I/O kernels must manage both streaming data storage, as well as sequential file retrieval.

SAR Processing
Layered Architecture for parallel computing

- Kernel layer does single-node math & parallel messaging
- Library layer provides a parallel data and computation toolbox to Matlab users via maps
- Good pseudo-code for expressing parallel constructs
Anatomy of a Map

Maps separate algorithm development from algorithm distribution

\[
\text{mapA} = \text{map}( [2 \ 2], \ {} , \ 0:3 );
\]

Grid specification together with processor list describe where the data is distributed.

Distribution specification describe how the data is distributed (default is block).

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

MATLAB constructors are overloaded to take a \text{map} as an argument, and return a \text{dmat}, a distributed array.
Outline

- Introduction
- **Parallel Strategies**
  - Coarse grained
  - Fine grained
  - Pipelined
- Results
- Summary
Computation has traditionally focused on Computation ...

... but Data I/O performance is increasingly important
Coarse Grained Parallelism
• Parallelize system mode
• End-to-End (computation and I/O components)
• Look for independence
• Stages, images, detection are independent

Process images independently:
no inter-processor communication

Caveats:
• Higher latency
• Fit on one processor
% Create Maps - distribute rows
map1 = map([Ncpus 1],{},0:Ncpus-1);

% Create z - distributed matrix.
Z = zeros(n, m, map1);

% Get the local indices
my_Z = local(Z);
my_ind = global_ind(Z,1);

for ilocal = 1:length(my_ind)
    iglobal = my_ind(ilocal);
    my_Z(ilocal,:) = userfunction(iglobal);
end

% Copy local portion to global
Z = put_local(Z, my_Z);

- Example illuminates use of distributed matrices
- Local-global mapping is abstracted away
- Parallelism is derived from locality
- Communication automatic via use of maps
SAR – Coarse Grained Implementation

Map Creation
mapImage = map([1 Ncpus],{},0:Ncpus-1);
GlobalImages = zeros(numImages,mapImage);
LocalImages = global_ind(GlobalImages);

Stage 1, Kernel 1 (Form Images)
for iImageLoop = 1:nLocalImages
    iImage = LocalImages(iImageLoop);
    readRawData;
    formImage(iImage);
    insertTemplates;

Stage 1, Kernel 2 – Image Storage
fwrite(image);

- Create map for images
- Images created independently
- Images written to disk independently
- Minimal code modification

Parallel File System
MIT Lincoln Laboratory
Outline

• Introduction
• Parallel Strategies
  – Coarse grained
  – Fine grained
  – Pipelined
• Results
• Summary
HPEC SAR Benchmark
Fine Grained Parallel Challenges

- Extract computational chain
- Focus on Stage 1, Kernel 1

- I/O larger than single processor memory
- FFT, IFFT
- Corner Turn (3)
- FFTShits

- Load balancing
- Large & small I/O

- Distributed Images
  - Differencing
  - Thresholding
- Distributed ROI
- Load balancing

Scalable Data and Template Generator
Kernel #1 Image Formation
Template Insertion
Kernel #4 Detection
Validation

Raw SAR
SAR Image
SAR Image
SAR Image
SAR Image
Templates
Templates
Templates
Templates

Kernel #4 Detection
SAR Image
SAR Image
SAR Image
SAR Image
Templates
Templates
Templates
Templates

Insertion

Scalable Data and Template Generator
Kernel #1 Image Formation
Template Insertion
Kernel #4 Detection
Validation

I/O larger than single processor memory
FFT, IFFT
Corner Turn (3)
FFTShits
Load balancing
Large & small I/O
Distributed Images
Differencing
Thresholding
Distributed ROI
Load balancing
Front-End: SAR Image Formation Stages

- Assumption: Input raw SAR data exceeds processor DRAM
- All stages can be block distributed
- Dominated by the corner turns (all-to-all communication)
Front-End: SAR Image Formation Maps

Map structures look like:

- **Columns:** \( \text{ColMap} = \text{map}([1 \ Np], \{\}, \ 0:0\Np-1); \)
- **Rows:** \( \text{RowMap} = \text{map}([\Np 1], \{\}, \ 0:0\Np-1); \)

Asynchronous File Input

Pulse Compression (FFT - columns)

Cross-range Re-sampling (FFT, IFFT)

Matched Filter & Interpolation (FFT rows)

Back-projection Image Conversion (1st half – FFT columns)

Image Conversion (2nd half – FFT rows)

Template Insertion

Image Storage

File Read

Corner Turn

File Write
Corner Turn: All-to-All

```c
my_rank=MPI_Comm_rank(comm);
if (my_rank==0)|(my_rank==1)|(my_rank==2)|(my_rank==3)
    Xlocal=rand(M,N/4);
endif
if (my_rank==4)|(my_rank==5)|(my_rank==6)|(my_rank==7)
    Zlocal=zeros(M/4,N);
endif
Xlocal=fft(Xlocal);
tag=0;
if (my_rank==0)|(my_rank==1)|(my_rank==2)|(my_rank==3)
    start=1;
    len=M/4;
    for dest_rank=4:7
        last=start+len-1;
        MPI_Send(dest_rank,tag,comm,Xlocal(start:last,:));
        start=last+1;
    end
endif
if (my_rank==4)|(my_rank==5)|(my_rank==6)| (my_rank==7)
    start=1;
    len=N/4;
    for recv_rank=0:3
        last=start+len-1;
        Zlocal(:,start:last)=MPI_Recv(recv_rank,tag,comm);
        start=last+1;
    end
endif
End
Zlocal=fft(Zlocal);
```

**MatlabMPI:**
- Efficient
- Complicated
- Inelegant

**pMatlab**
- Elegant
- 2x slower

**Transpose_grid**
- Elegant
- Optimized messaging
- Efficient

- Transpose_Grid balances trade-off between efficiency and simplicity

- X = fft(X,[],2);
  Z(:, :) = X;
- Z = transpose_grid(X);
- Z = fft(Z);
- X = fft(X);
- Z = transpose_grid(X);
- Z = fft(Z);
pMatlab 2D FFT Code – Pipelined
8 Processor Example

1. \( Np = \text{pMATLAB.comm\_size}; \) % Set number of processors.
2. \( P = 2^{10}; \quad Q = 2^{10}; \) % Set dimensions of array.
3. \( Zmap = \text{map}([1 \ 4],\{},4:Np-1); \) % Row map – processor set 1
4. \( Xmap = \text{map}([4 \ 1],\{},0:3); \) % Column map – processor set 2

% Create complex global arrays X and Z for FFT.

5. \( X = \text{complex}(\text{rand}(P,Q,Xmap),\text{rand}(P,Q,Xmap)); \)
6. \( Z = \text{complex}(\text{zeros}(P,Q,Zmap)); \)
7. \( X = \text{fft}(X,[],2); \) % FFT rows.
8. \( Z(:,:,1) = X; \) % Redistribute data.
9. \( Z(:,:,1) = \text{fft}(Z,[],1); \) % FFT columns.

- Maps use different processors sets
- Two line change – map creation
- Remaining code is unchanged
- No double buffering
Outline

• Introduction
• Parallel Strategies
  – Coarse grained
  – Fine grained
  – Pipelined
• Results
• Summary
Lincoln Laboratory Grid
LLGrid

Service Nodes
- Network Storage
- Resource Manager
- Login Nodes
- Cluster Management
- Cluster Monitoring

Compute Nodes
- Cluster Switch

LLAN Switch

- 280 processors (900+ processors, soon)
- Gigabit Ethernet
- 1 Terabyte of storage – growing to 36 Terabytes
- Standard Parallel Software

<table>
<thead>
<tr>
<th></th>
<th>Dell PowerEdge 2650</th>
<th>Dell PowerEdge 1750</th>
<th>Dell PowerEdge 1855MC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUs</td>
<td>64 CPUs (32 nodes)</td>
<td>96 CPUs (48 nodes)</td>
<td>300 CPUs (150 nodes, 10 chassis)</td>
</tr>
<tr>
<td>Xeon</td>
<td>2.8 GHz Xeon (32-bit)</td>
<td>3.06 GHz Xeon (32-bit)</td>
<td>3.2 GHz EM64T Xeon (64-bit)</td>
</tr>
<tr>
<td>RAM</td>
<td>4 GB RAM</td>
<td>6 GB RAM</td>
<td></td>
</tr>
<tr>
<td>Ethernet</td>
<td>Two Gigabit Ethernet interfaces</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS</td>
<td>Red Hat Enterprise Linux 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIT Lincoln Laboratory
Coarse Grained Speedup Results
End-to-End

End-to-End File I/O and Computation Performance
for 2, 4, 8, 16, 32, and 64 processors

SCALE=12, Image Size = 3K x 4.5 K pixels, 100 Images

- End-to-end compute performance initially linear then falls off
- End-to-end I/O initially linear decays as system saturates

Storage System Saturation
Coarse Grained Speedup Results
Computational Kernels

Speedup for 2, 4, 8, 16, 32, and 64 processors

SCALE=12, Image Size = 3K x 4.5 K pixels, 100 Images

- Image Formation nearly linear performance
- Target Detection victim of load imbalance

Increasing number of images
Corner Turn Performance

- MatlabMPI has slightly higher efficiency than Transpose_Grid.
- Both Transpose_Grid and MatlabMPI achieve higher efficiency than pMatlab.
- Transpose_Grid provides efficiency and simplicity.
Summary

- Coarse Grained Strategy yields linear speed-up
- Approximately 30 lines of new code were added to 1400 lines of SSCA#3
- Evaluation of Corner Turn performance indicates that Transpose_Grid provides efficiency and elegance
- Maps provide a means for creating fine grained parallel process chain
- Use of maps in pipelined corner turn requires minimal code changes

Future Work
- Fine Grained Implementation
- Pipelined Implementation
Acknowledgements

• Lincoln Laboratory pMatlab Team
  – Nadya Bliss
  – Hahn Kim
  – Albert Reuther

• Sponsor
  – DARPA HPCS Program