An Evaluation of Parallelization Techniques for MRF Image Segmentation

Shane Mottishaw, Sergey Zhuravlev, Lisa Tang, Alexandra Fedorova, and Ghassan Hamarneh

Presented by: Ahmed Saad

Medical Image Analysis Lab,
School of Computing Science, Simon Fraser University, Canada

smottish@sfu.ca
Motivation

- MRFs are of great interest to the medical image community
  - Captures spatial information
  - Many successful applications
- Problems:
  - Parameter selection
  - Computational complexity
Motivation

• Solution: acceleration via parallelization
  • Mitigates computational complexity
  • Enables real-time interaction
Evaluation

• Chose Belief Propagation (BP) MRF optimization

• Parallelize for multi-core processors

• Why not GPU?

  • Lack of global synchronization

  • Programming model can reduce productivity and performance
BP Overview

- Image represented as a graph
  - Nodes represent pixels
  - Edges defined by neighbourhood
- Nodes pass messages to their neighbours
  - Node $p$ passes message $m_{pq}$ to node $q$
- After $T$ iterations, accumulated messages determine labeling
BP Messages

- In iteration $t$, node $p$ sends message $m_{pq}^{t}$ to node $q$:

$$m_{pq}^{t}(f_q) = \min_{f_p} \left( V(f_p, f_q) + D_p(f_p) + \sum_{s \in N(p) \setminus q} m_{sp}^{t-1}(f_p) \right)$$

  - smoothness term
  - data term
  - received messages from $t-1$

- $m_{pq}^{t}$ is a vector with size = number of labels
BP Labelling

• After $T$ iterations, a pixel’s label $b_q$ is calculated as:

$$b_q(f^*_q) = D_q(f^*_q) + \sum_{p \in N(q)} m^t_{pq}(f^*_q)$$

• Where $f^*_q$ is the optimal label.
Parallelizing BP: Dependencies

- If two sections/blocks of code share heap data, **data dependency** exists.
- In BP, neighbours will read/write shared memory to send messages.
- There is also a dependency between iterations.

\[
\text{msg}[q][p] = \text{calc}(\text{msg}[p][q])
\]

\[
\text{msg}[p][q] = \text{calc}(\text{msg}[q][p])
\]
Parallelizing BP: Inter-processor communication

• Concurrent shared memory access means:
  • Inter-processor/inter-thread communication, and/or
  • Synchronization

• Communication increases latencies and contention

• Must carefully manage data and synchronization!
Implementation

• “Typical” implementation
  • Neighbours share a queue to send/receive messages
  • Problem: requires synchronization for each read/write!
• Double buffering
  • Separate read/write or input/output queues
  • Single synchronization step to “swap”
Implementation

• In iteration t, synchronization required to guarantee that:
  
  • A node has received all its messages from t-1

  • A node only sends to neighbour if neighbour also in t
Implementation: Partitioning

- Each thread works on an equal partition of nodes

- Each thread is bound to a core

- When message sent from node in partition X to node in partition Y, inter-thread/inter-core communication

- Assign “neighbouring partitions” to threads/cores on same chip
Implementation: Message Passing

Algorithm 1: parallel message-passing routine executed by each thread

Input: number of iterations $T$ and a set of nodes $N$

begin
for $t \leftarrow 1$ to $T$ do
for each $p \in N$ do
foreach $q \in \text{neighbours}(p)$ do
message $\leftarrow m_{pq}^t$
while $\text{stateOf}(q) \neq \text{RESET}$ do
end
sendMsg(message)
end
while $\text{stateOf}(p) \neq \text{FULL}$ do
end
swapBuffers($p$)
stateOf($p$) $\leftarrow \text{RESET}$
end
end
Implementation: “lock-step”

**Algorithm 2:** parallel lock-step routine executed by each thread

1. begin
2. for $t \leftarrow 1$ to $T$ do
3.     foreach $p \in N$ do
4.         foreach $q \in \text{neighbours}(p)$ do
5.             $m_{pq}^t \leftarrow$ message
6.             sendMsg(message)
7.         end
8.     end
9.     spinWait()
10.    swapBuffers(p)
11.    spinWait()
12. end
13. end
Experimental Setup

- Binary image segmentation
  - 600x450 greyscale image
  - 15 iterations

- Measure the time it takes for all threads to complete an iteration

- Run-time = sum of iteration times

- Averaged over 10 trials
Experimental Setup: Machines

2x 4-core Intel Xeon E5405

core 0  
L1  

core 1  
L1  

core 2  
L1  

core 3  
L1  

L2  

L2  

4x 6-core AMD Opteron 2435

core 0  
L1/L2  

core 5  
L1/L2  

L3  

L3  

Module 0  

Module 1  

smottish@sfu.ca
Results: “Message Passing”

- Run-times

![Graph showing run-times for 2x4-Core Intel Xeon and 4x6-Core AMD Opteron (NUMA)]
Results: “Lock-step”

- Run-times

2x4-Core Intel Xeon

<table>
<thead>
<tr>
<th>Threads</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

4x6-Core AMD Opteron (NUMA)

<table>
<thead>
<tr>
<th>Threads</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
</tr>
</tbody>
</table>
Results: “Message Passing”

- Speed-up

![Graph showing speed-up for 2x4-Core Intel Xeon and 4x6-Core AMD Opteron (NUMA) with varying threads.](image)
Results: “Lock-step”

- Speed-up

**2x4-Core Intel Xeon**

**4x6-Core AMD Opteron (NUMA)**

---

smottish@sfu.ca
Conclusion

• Achieved a maximum speedup of 8.08

• Managing data and inter-processor communication is crucial!

• Further optimization can be done to further reduce:
  • Memory latencies
  • Inter-processor communication

• Implement and compare a GPU version

• Hybrid CPU-GPU version