An Evaluation of Parallelization Techniques for MRF Image Segmentation

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

Simon Fraser University

Abstract. Markov Random Fields (MRFs) are of great interest to the medical image analysis community but suffer from high computational complexity and difficulties in parameter selection. For these reasons, efforts have been made to develop more efficient algorithms for solving MRF optimization problems in order to enable reduced run-times and better interactivity. However, these algorithms are often implemented in serial and thus do not benefit from multi-core technology. In this work, we demonstrate a parallelized implementation of a popular MRF optimization algorithm, belief propagation, and use it to perform a binary image segmentation. By utilizing modern, lightweight parallel-programming techniques we are able to achieve a speedup of approximately 8 times, reducing the average segmentation time of a single 600×450 image from 12.7s to 1.6s.

1 Introduction

Markov Random Field (MRF) optimization is an energy minimizing framework, which models image data as graphs, captures spatial interaction between image pixels through the graph's connectivity, and reformulates medical image computing tasks as graph labelling problems. Since the seminal work of Geman and Geman [8], MRFs have had successful applications in many different image analysis problems [11, 6] including segmentation [4], registration [18], image restoration [3] and stereo matching [5]. Because of these successes, MRFs are of great interest to the medical image analysis community. However, one of the difficulties with MRF models is the proper selection of parameters [10]. This makes it difficult to provide automatic solutions and thus interactive methods are used in order to increase effectiveness [4]. However, interactive methods require the user to modify parameters and then decide whether the updated segmentation is correct or requires further input, therefore interactive methods must execute in near real-time to be efficient. Another problem in MRF theory is that large neighbourhoods and/or multiple labels make finding the global optimum of the energy function to be minimized an NP-hard problem and is therefore intractable [11]. This leads to approximation solutions that operate in polynomial time, but even these methods are computationally intensive [7]. The computationally intensive nature of MRF optimization is further compounded

by continuous advances in medical imaging leading to ever increasing image resolutions and sizes, making execution times even longer. Due to the wide range of successful applications, combined with the need for increased efficiency and effectiveness, it is important to accelerate the computation involved in solving MRF energy minimization problems.

Efforts have already been made to develop more efficient MRF optimization algorithms [15]. Of these algorithms, belief propagation (BP) has become a popular algorithm due to its ability to find local minima over "large neighbourhoods", while maintaining reasonable computational complexity [17, 15, 7]. However, despite initial advances in the form of algorithms such as BP, MRF formulations remain computationally expensive and researchers continue to develop algorithmic techniques to improve running times [7].

The recent transition from single-core to multi-core processor architectures provides new opportunities for accelerating medical image analysis algorithms. However, in order to realize these potential gains, parallel software is required. Despite the reduced runtimes of algorithms such as BP, these algorithms are still serial and thus do not benefit from increased parallelism in hardware. In fact, as the number of cores increases, these algorithms may actually suffer. This is because, for some architectures, the amount of power provided to the chip is decreased to allow multiple cores to be placed on the chip without exceeding power and heat constraints [14]. However, decreasing power also decreases the frequency at which each core can operate, leading to degradation in serial performance. This further necessitates the development of parallelized implementations of computationally expensive algorithms such as BP.

In this work, we focus on developing a parallel implementation of the BP algorithm that targets multi-core processors. Note that parallelism can be achieved by leveraging architectures other than multi-core processors. A primary example is graphics processing units (GPUs). GPUs have a highly parallel architecture in that they have many simple processing units dedicated to performing the highly parallel arithmetic tasks found in computer graphics [1]. This is in contrast to multi-core processors which currently have a relatively small number of more complex cores that are capable of performing a number of non-computational tasks such as branch prediction and prefetching. The highly parallel architecture of GPUs seems to make them an obvious choice for parallelization, however their specialized nature introduce a number of inherent problems that have motivated us to implement a parallel solution for multi-core processors instead. Despite the fact that GPUs have become increasingly general-purpose with the advent of technologies such as NVIDIA's Compute Unified Device Architecture (CUDA) [12], there still exist applications for which GPUs are not well suited [1]. For example, GPUs are still limited to certain types of parallelism due to the lack of support for global synchronization [13]. Even when the available parallelism is amenable to GPUs, the specific challenges and unfamiliar programming model inherent to GPUs increase programming complexity [1] which can lead to implementations that achieve only limited performance.

The primary focus of this paper is to demonstrate the use of modern parallel programming techniques for accelerating MRF based image segmentation. We provide a serial and parallel implementation of the max-product BP algorithm outlined in [7] and discuss the challenges involved when parallelizing this algorithm for modern multi-core architectures.

2 Belief Propagation for MRF Optimization

In medical image analysis, MRF formulations are classically defined in the framework of a pixel labelling problem, where each pixel $p \in \mathcal{P}$ is assigned a label $f_p \in \mathcal{L}$. The quality of a given labelling is measured by the energy function

$$E(f) = \sum_{(p,q)\in\mathcal{N}} V(f_p, f_q) + \sum_{p\in\mathcal{P}} D_p(f_p) \tag{1}$$

where $\mathcal{N}$ defines an unordered set of neighbouring pixels. $V(f_p, f_q)$, the smoothness term, is the cost of assigning the labels f_p and f_q to neighbouring pixels p and q. $D_p(f_p)$, the data term, is the cost of assigning pixel p the label f_p. By minimizing the function in equation (1), a labelling is found which corresponds to the maximum a posteriori probability (MAP) estimate. In this paper, we focus on the BP optimization algorithm for solving this minimization problem.

BP represents an image as a graph of nodes with edges defined by a neighbourhood system (e.g. 4 or 8 pixel connectivity for 2D). In each iteration of the algorithm, nodes pass messages to their neighbours. Messages are vectors of size M, where M is the number of labels $|\mathcal{L}|$. A message m^t_{pq} from node p to q on iteration t is calculated using the following equation for each possible label f_p:

$$m^t_{pq}(f_q) = \min_{f_p} \left(V(f_p, f_q) + D_p(f_p) + \sum_{s\in\mathcal{N}(p)\setminus q} m^{t-1}_{sp}(f_p) \right) \tag{2}$$

where m^{t-1}_{sp} is a message sent to p from neighbour s during the previous iteration. Note that a message contains information about smoothness costs, data-costs and previously received messages. At any point, the current labelling is defined by:

$$b_q(f^*_q) = D_q(f^*_q) + \sum_{p\in\mathcal{N}(q)} m^t_{pq}(f^*_q) \tag{3}$$

where f^*_q is the label that minimizes (3) for node q. The BP algorithm runs for a fixed number of iterations T and the resulting labelling defined by $b_q(f^*_q)$ in (3) represents the final segmentation.

3 Parallelizing BP

3.1 BP Algorithm Dependencies

Before an algorithm can be parallelized, its dependencies must be identified and analyzed. When sections of code (either identical or distinct) can be executed

on different threads and share global data, or depend on one section of code to be completed before another can proceed, a dependency exists between these sections. In BP, it is clear that nodes will need to share memory in order to "send" and "receive" messages, and thus a dependency exists between neighbouring nodes. Specifically, the calculation of messages m_{pq} requires access to messages sent by its neighbours in the previous iteration. This also means that there is a dependency between iterations, since a node must receive all its messages before it can calculate messages for the next iteration.

When dependencies exist, inter-thread communication and/or synchronization is required to enforce ordering or to protect concurrent accesses to shared memory in order to avoid race conditions. Inter-thread communication and synchronization is achieved through shared memory. Multi-core, also referred to as chip-multiprocessing (CMP) technology, has turned even commodity machines into shared memory multiprocessors (SMP), where each processing core shares one or more memory domain. SMP enables communication and synchronization between threads in a parallel application. However, while a program may view shared memory as a single shared space, the underlying architecture is more distributed. Cores on a chip will often have their own private cache (L1 and sometimes L2) and will also share a common cache memory (either L2 or L3). In a simple case, all cores will share a single memory module (main memory) but in some cases, such as with Non-Uniform Memory Architectures (NUMA) each chip may have its own local memory module, but can still access memory located in other modules. Therefore with SMP, there can be many copies of shared data located in different caches or modules.

In order to provide the SMP abstraction, the hardware keeps track of these copies through a cache coherency protocol. In these protocols, when a core modifies a shared line in cache, it must send out an invalidation message so that other cores know that their copy is no longer valid. When a core attempts to read an invalid cache line, it will have to either fetch it from another cache, or from main memory, incurring an increased memory access latency. Inter-thread communication and synchronization therefore not only causes increased memory latencies, but also increased bus traffic, causing contention over shared busses and memory controllers. We address the importance of the careful management of data and synchronization in our parallel implementation, which is discussed in detail in section 3.2.

3.2 Implementation

A natural implementation of BP would likely provide each pair of neighbour nodes with a single queue that they can read from and write to. However, this would require synchronization to protect each read and write to the shared queue. A more efficient implementation would be to use double buffering: during an iteration, nodes read from an "input" buffer that stores previous message and write to an "output" buffer. These reads and writes are exclusive: only one node ever reads from/writes to a queue during the iteration. This allows for messages to be computed and sent independently, and thus concurrently, by each node

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

```
   Input: number of iterations T and a set of nodes N
 1 begin
 2     for t ← 1 to T do
 3         foreach p ∈ N do
 4             foreach q ∈ neighbours(p) do
 5                 message ← m_pq^t
 6                 while stateOf(q) ≠ RESET do
                       ▷loop until condition met i.e. spin
 7                 end
 8                 sendMsg(message)
 9             end
10             while stateOf(p) ≠ FULL do
                   ▷loop until until condition met i.e. spin
11             end
12             swapBuffers(p)
13             stateOf(p) ← RESET
14         end
15     end
16 end
```

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

```
   Input: number of iterations T and a set of nodes N
 1 begin
 2     for t ← 1 to T do
 3         foreach p ∈ N do
 4             foreach q ∈ neighbours(p) do
 5                 message ← m_pq^t
 6                 sendMsg(message)
 7             end
 8         end
 9         spinWait()
10         swapBuffers(p)
11         spinWait()
12     end
13 end
```

Algorithm 3: spinWait synchronization method for lock-step

```
 1 begin
 2     tid ← getThreadID()
 3     activeThreads ← atomic_dec(sharedCounter)
 4     if activeThreads > 0 then
 5         while stateOfThread(tid) = WAIT do
               ▷loop until condition met i.e. spin
 6         end
 7     else
 8         activeThreads ← numThreads
 9         for i ← 1 to numThreads do
10             stateOfThread(tid) ←!WAIT
11         end
12     end
13     stateOfThread(tid) ← WAIT
14 end
```

within an iteration. Note that while two nodes still share memory (namely, each of the buffers), only a small amount of synchronization is used to "swap" these buffers, as opposed to synchronizing each read and write. Once the buffers have been "swapped", a node will then read newly received messages in the "output" buffer, and write over old messages in the "input" buffer.

Synchronization is required to guarantee two conditions: 1) that a node has received all its messages, and 2) that a node only sends to a neighbour if that neighbour is in the same iteration as the sender (i.e. has already "swapped" its buffers). Note that without condition 2), swaps could occur during or after a node sends messages to its neighbour without the neighbour ever reading these messages, thus causing messages to be "lost".

The dependencies outlined in section 3.1 led us to implement two different parallelization schemes. Our first implementation, which we call "message-passing", is outlined in algorithm 1. In this implementation, nodes "wait" (i.e. spin on a shared variable) until they have received all messages. Also, sender nodes "wait" until neighbours are ready to receive. This essentially transforms the BP algorithm into a parallel message passing system where each node sends T messages in parallel, based on the above rules. Theoretically, this will also allow some nodes to proceed to the next iteration, while other nodes are completing previous iterations; this is referred to as pipeline parallelism. However, as it will be shown in the next section, this led to limited performance gains. Therefore, we implemented another, simpler parallelization scheme where nodes still send messages in parallel but must wait for all other nodes to send/receive messages and then must wait again for all nodes to swap their buffers before continuing to the next round of message passing. In this way, nodes pass messages and update their assigned nodes' buffers in lock-step. This method, called "lock-step", is outlined in algorithm 2. Algorithm 3 shows the pseudo-code for the `spinWait` synchronization method used in algorithm 2.

In both schemes, each thread is given an equal partition of the nodes (denoted by N in algorithms 1 and 2) in the graph for BP. Each thread is bound to a core and calculates and sends messages for each node in its partition. Note that this will necessarily cause inter-thread communication when a message is sent from a node in one partition to another. This is because a send is a write operation into another node's buffer which causes an invalidation and since a node will read from this buffer in the next iteration, it must fetch the new value from another cache, or from main memory. The effects of this communication is mitigated by placing threads with adjacent partitions on the same chip. This essentially allows threads to communicate via the last level cache on the chip, reducing bus contention and memory latencies as discussed in section 3.1.

4 Results

To calculate the speedup achieved by our parallel implementations of BP, we performed a binary segmentation using a four-connected grid of the 600×450 image in figure 1. The image also contains brush strokes denoting background

Fig. 1. 600×450 input image (left) and resulting MRF BP segmentation (right)

and foreground regions and was converted to greyscale before segmentation. During execution, we measured the start and stop times of each thread for each iteration, where an iteration consisted of calculating and sending messages for each node assigned to a thread. By taking the difference between the earliest start time and latest stop time, we get an accurate measurement of the parallel computation of a single iteration. We then calculate the total run-time as the sum of measurements for each iteration, for a sufficient number of iterations (15 for our experiments).

We ran this experiment on two different machines. The first machine has two Quad-Core Intel Xeon E5405 processors for a total of 8 cores. On each chip, two cores share a 6 megabyte L2 cache giving a total of 12 megabytes per chip and 24 megabytes overall. The second machine has four Six-Core AMD Opteron 2435 processors for a total of 24 cores. All cores on each chip share a 6 megabyte L3 cache for a total of 24 megabytes across all processors.

The resulting binary segmentation of these images can be seen in Figure 1. While the quality of the segmentation is not the focus of this paper, to demonstrate the effectiveness of our BP implementation, we have calculated the Dice Similarity Coefficient (DSC) [2] between the segmentation in figure 1 and the global minimum solution computed by tree-weighted message passing [16, 9] to be 0.997. We also calculated the DSC between a serial segmentation and a parallel segmentation (for both "message passing" and "lock-step") to be 1.0, which confirms the correctness of our parallel implementations.

Figure 2 shows the results for the "message passing" scheme. Figures 2(a) and 2(b) show the total run-time (y-axis) averaged over 10 trials for varying numbers of threads (x-axis). Figures 2(c) and 2(d) show the resulting speedup (y-axis) achieved for different numbers of threads (x-axis) where speedup is the number of times faster a computation runs, calculated as

$$speedup = Time_{serial}/Time_{parallel} \tag{4}$$

where $Time_{serial}$ is the run-time (as described above) for the serial, or single threaded case and $Time_{parallel}$ is the run-time for parallel case with n threads. Figure 3 shows the same results for the "lock-step" scheme. Immediately, we see that the "lock-step" scheme performs significantly better. Further analysis is required to verify the reason for this discrepancy in performance, but we provide a possible explanation for this discrepancy.

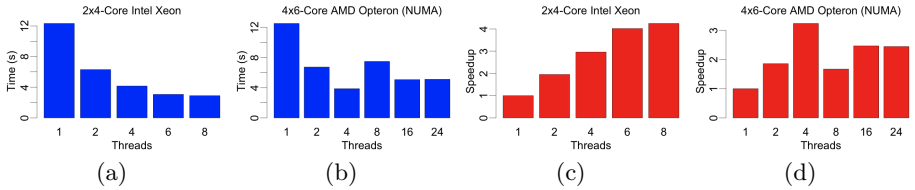

Fig. 2. BP "message-passing" run-times and speedups for 8 and 24 core machines

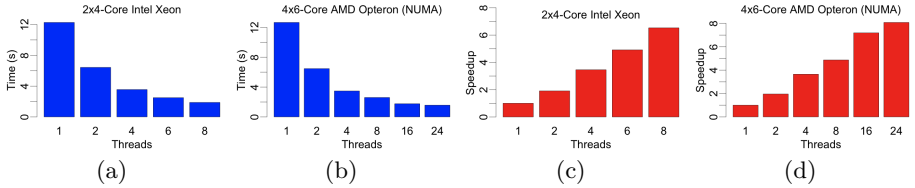

Fig. 3. BP "lock-step" run-times and speedups for 8 and 24 core machines

In the "message passing" scheme, each node has up to four synchronization variables, one for each of its "input"/"output" double-buffers (since there exists a double-buffer between each pair of neighbouring pixels). These variables are used to make nodes "wait" to receive and send messages. Since these variables are written to each time buffers are swapped, an invalidation will occur for each swap, which may cause increased memory access latencies when these variables are subsequently read. This also means that the number of invalidations is proportional to the number of nodes M. In the "lock-step" scheme, there is only one synchronization variable per-thread (used for signalling threads), plus a globally shared counter. Since each thread only decrements the counter once per iteration, the number of invalidations is proportional to the number of threads N. Since $N << M$, there are significantly fewer invalidations, and thus there is significantly reduced inter-core communication and memory access latencies.

These results also demonstrate a pervasive problem in parallel computing. Notice that even with "lock-step", the marginal gain from increasing threads starts to decrease at higher numbers of threads. This is due to both inter-core communication as already described, as well as contention over shared resources, as described in section 3.1. In BP, writing to shared buffers causes inter-core communication through invalidation messages and also increases memory latencies and bus traffic from cores having to access data from other cores or other chips. This becomes even more of a problem for non-uniform memory architectures (NUMA), as is the case with the AMD machine. On these machines, not all cores access the same memory bank. Instead, each chip accesses its own memory domain, and if it requires data that is not in its local domain, it must access it from another one, incurring an added cost. When inter-core communication translates to data being fetched from other domains, performance can suffer greatly. Assigning threads that communicate with each other to the same core/-

domain mitigates this issue, but does not solve it. Also, the cache footprint for BP is large; each node data structure is 576 bytes (as determined by the `sizeof` compile-time operator), and with 600×450 nodes, this gives a total of about 148 MB. This means that main memory will be accessed frequently which increases contention over shared memory busses and controllers. Despite these issues, we realized a maximum speedup of 6.53 on the Intel machine, and 8.08 on the AMD machine.

5 Conclusions

We have demonstrated how modern techniques in parallel computing can be used to accelerate the popular BP MRF optimization algorithm. Our results demonstrate that significant speedup can be achieved without compromising the accuracy of the algorithm or making extra assumptions, as is often done with algorithmic optimizations. However, this speedup is only achieved by carefully managing inter-thread communication/synchronization and by taking into account cache utilization and shared resource contention. In our case, we were able to achieve a maximum speedup of 8.08.

Despite these speedups, work can still be done to further optimize our parallel implementation of BP by continuing to reduce memory latencies and inter-core communication. For example, memory latencies can be hidden by manually prefetching data. Before performing calculations for a current node, we can prefetch the data required for the calculation of the next node. This way, we try to guarantee that the required data is always in cache. Reducing the cache footprint of the algorithm and efficiently packing data will also increase the chance that necessary data is in cache, thus decreasing memory latencies. For inter-core communication, we can look for places where false sharing is occurring and can pad and align data types/structures accordingly. False sharing occurs when two threads share some data on a cache line, but not all data. When this cache line is invalidated, data not shared by threads is also invalidated. By ensuring that only shared data is on the same cache line, only the shared data is invalidated, leaving the non-shared data in cache for thread/core local computations.

Our parallel implementation of BP is also a potential candidate for conversion to a GPU implementation. In a GPU implementation, instead of having nodes send messages right away, we can change the message passing protocol into a two phase program, where in the first phase, all nodes calculate all their messages, and in the second phase, these results are written to the appropriate buffers. The first phase is a massively data parallel operation that would be well suited for a GPU (which can have up to 512 processing units) capable of processing nodes in parallel. The results of this would be sent back to the CPU where the appropriate synchronization can be done to ensure buffers are updated properly.

6 Acknowledgements

This project incorporated code written by the authors in [15].

References

1. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.
2. M. Bach Cuadra, L. Cammoun, T. Butz, O. Cuisenaire, and J. Thiran. Comparison and validation of tissue modelization and statistical classification methods in T1-weighted MR brain images. 2005.
3. J. Besag. On the statistical analysis of dirty pictures. *Journal of the Royal Statistical Society*, B-48:259–302, 1986.
4. Y. Boykov and M.P. Jolly. Interactive graph cuts for optimal boundary and region segmentation of objects in N-D images. *Proc. Eighth IEEE Intl Conf. Computer Vision*, 1:105–112, 2001.
5. Yuri Boykov, Olga Veksler, and Ramin Zabih. Fast approximate energy minimization via graph cuts. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23(11):1222–1239, 2001.
6. R. Chellappa and A. Jain. *Markov Random Fields: Theory and Applications*. Academic Press, 1993.
7. Pedro Felzenszwalb and Daniel Huttenlocher. Efficient belief propagation for early vision. *International Journal of Computer Vision*, 70(1):41–54, 2006.
8. Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distributions and the bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(6):721–741, November 1984.
9. Vladimir Kolmogorov. Convergent tree-reweighted message passing for energy minimization. *IEEE Trans. Pattern Anal. Mach. Intell.*, 28(10):1568–1583, 2006.
10. S. Li. Markov random field models in computer vision. *Computer Vision ECCV '94*, pages 361–370, 1994.
11. S.Z. Li. *Markov Random Field Modeling in Image Analysis*. Tokyo: Springer, 2001.
12. Paulius Micikevicius. CUDA: The democratization of parallel computing. http://developer.download.nvidia.com/presentations /2008/SIGGRAPH/S2008_CUDA_Paulius.pdf, 2008.
13. Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Princ. and Practice of Parallel Prog.*, pages 73–82, 2008.
14. Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.
15. Richard Szeliski, Ramin Zabih, Daniel Scharstein, Olga Veksler, Vladimir Kolmogorov, Aseem Agarwala, Marshall Tappen, and Carsten Rother. A comparative study of energy minimization methods for markov random fields with smoothness-based priors. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30:1068–1080, 2008.
16. Martin Wainwright, Tommi Jaakkola, and Alan Willsky. MAP estimation via agreement on (hyper)trees: Message-passing and linear programming approaches. *IEEE Transactions on Information Theory*, 51:3697–3717, 2002.
17. J. Wu and A. C. Chung. Cross entropy: a new solver for markov random field modeling and applications to medical image segmentation. *MICCAI 2005*, 8(1):229–237, 2005.
18. Paul Wyatt and J. Noble. MAP MRF joint segmentation and registration. *MICCAI 2002*, pages 580–587, 2002.