Lawrence Berkeley National Laboratory
LBL Publications

Title

Shared-Memory Parallel Probabilistic Graphical Modeling Optimization: Comparison of Threads, OpenMP, and Data-Parallel Primitives

Permalink
https://escholarship.org/uc/item/3z48p6kq

ISBN
9783030507428

Authors
Perciano, Talita
Heinemann, Colleen
Camp, David
et al.

Publication Date
2020

DOI
10.1007/978-3-030-50743-5_7

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Shared-Memory Parallel Probabilistic Graphical Modeling Optimization: Comparison of Threads, OpenMP, and Data-Parallel Primitives

Talita Perciano[1(✉)], Colleen Heinemann[1,2], David Camp[1], Brenton Lessley[3], and E. Wes Bethel[1]

[1] Lawrence Berkeley National Laboratory, Berkeley, USA
{tperciano,dcamp,ewbethel}@lbl.gov
[2] University of Illinois at Urbana-Champaign, Champaign, USA
heinmnn2@illinois.edu
[3] Verb Surgical, Inc., Santa Clara, USA

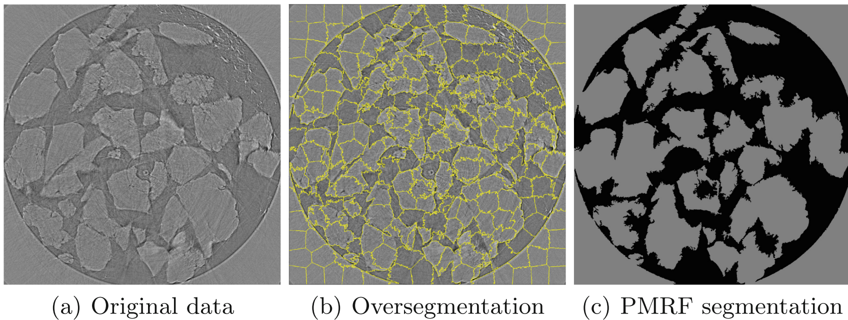
Abstract. This work examines performance characteristics of multiple shared-memory implementations of a probabilistic graphical modeling (PGM) optimization code, which forms the basis for an advanced, state-of-the art image segmentation method. The work is motivated by the need to accelerate scientific image analysis pipelines in use by experimental science, such as at x-ray light sources, and is motivated by the need for platform-portable codes that perform well across many different computational architectures. The primary focus of this work and its main contribution is an in-depth study of shared-memory parallel performance of different implementations, which include those using alternative parallelization approaches such as C11-threads, OpenMP, and data parallel primitives (DPPs). Our results show that, for this complex data-intensive algorithm, the DPP implementation exhibits better runtime performance, but also exhibits less favorable scaling characteristics than the C11-threads and OpenMP counterparts. Based upon a set of experiments that collect hardware performance counters on multiple platforms, the reason for the runtime performance difference appears to be due primarily to algorithmic efficiency gains: the reformulation from the traditional C11-threads and OpenMP expression of the solution into that of data parallel primitives results in significantly fewer instructions being executed. This study is the first of its type to do performance analysis using hardware counters for comparing methods based on VTK-m-based data-parallel primitives with those based on more traditional OpenMP or threads-based parallelism. It is timely, as there is increasing awareness of the need for platform portability in light of increasing node-level parallelism and increasing device heterogeneity.

Keywords: Probabilistic graphical models · Modeling optimization · Markov random fields · Image segmentation · Computer vision · Data parallel primitives · Shared-memory parallel · Platform portability

© Springer Nature Switzerland AG 2020
P. Sadayappan et al. (Eds.): ISC High Performance 2020, LNCS 12151, pp. 127–145, 2020.
https://doi.org/10.1007/978-3-030-50743-5_7

1 Introduction

Image segmentation is a computationally intensive task, influencing scientific analysis pipelines as a critical element, particularly those that work with large image-based data obtained by experiments and advanced instruments, such as the X-ray imaging devices located at the Advanced Light Source at Berkeley Lab[1]. As such instruments continually update in spatial and spectral resolution, there is an increasing need for high-throughput processing of large collections of 2D and 3D image data for use in time-critical activities such as experiment optimization and tuning [3]. Our work here is motivated by the need for image analysis tools that perform well on modern platforms, and that are expected to be portable to next-generation hardware (Fig. 1).



(a) Original data (b) Oversegmentation (c) PMRF segmentation

Fig. 1. Going from a raw image obtained by experiment to a segmented image suitable for quantitative analysis involves multiple processing stages. This example shows a single 2D slice from a 3D image stack obtained by x-ray microscopy at the Advanced Light Source. Here, the original data (left) undergoes an image oversegmentation to produce coarse regions (middle), which then undergo an additional processing stage to produce a highly accurate segmentation (right). The focus of this paper is on the method for the final processing stage, which uses a probabilistic graphical model that is optimized using a Markov Random Field formulation.

This work centers on evaluating the viability of specific approaches for achieving platform portability and performance on multi- and many-core platforms. We focus on a specific data-intensive problem for this study known as probabilistic graphical model (PGM) optimization using a Markov Random Field (MRF) formulation to tackle image segmentation problems. Such methods are known for their high degree of accuracy, and, thanks to recent advances, their amenability to parallelization. We focus on shared-memory parallel performance in this study, with an eye towards future hybrid-parallel implementations that build on our previous works [11,17,18,30].

We parallelize this unsupervised, graph-based learning method applied to scientific image segmentation using three different approaches, and perform an

[1] Advanced Light Source website: http://als.lbl.gov/.

in-depth performance analysis of each of the three: first by using C11-threads, then using the Open Multi-Processing (OpenMP) API, and finally using data-parallel primitives (`Sort`, `Scan`, `Reduce`, etc.). The C11-threads implementation is the most coarse in terms of workload decomposition, where N pixel neighborhoods are spread across P threads, where each thread receives N/P of the work. The OpenMP implementation uses loop parallelization over the N neighborhoods, which is a finer-grained distribution than the C11-threads version, and also benefits from better load balance due to OpenMP's dynamic scheduling capabilities. The DPP implementation is the finest level of workload decomposition, where the K operations in a given DPP are divided in chunks of size C across P execution threads.

In this work, we evaluate the key performance characteristics of each implementation using strong scalability measures and runtime performance. We also analyze the factors that lead to these performance characteristics by examining hardware performance counters for metrics like code vectorization, number of instructions executed, and memory cache utilization. The main contributions of this paper are: (1) to compare performance of a PGM optimization algorithm implemented with VTK-m-based data parallel primitives with ones based on explicit threading and OpenMP; (2) to give insight into performance characteristics of PGM optimization using VTK-m-based data parallel primitives; (3) the first use of hardware performance counters to examine the performance of a VTK-m-based code, where previous works looking at visualization and rendering measure and report runtime only (e.g., [15–17,19,23,28]).

2 Background and Previous Work

In the following sections, we summarize works relating to image segmentation, graph-based methods including MRF, and approaches for performance and portability using C11-threads, OpenMP, and data parallel primitives.

2.1 MRF-Based Image Segmentation

The process of segmenting an image involves separating various phases or components from a picture using photometric information and/or relationships between pixels/regions representing a scene. This essential step in an image analysis pipeline has been given great attention recently when studying experimental data [29]. There are several different types of image segmentation algorithms, which can be divided into categories, such as: threshold-based, region-based, edge-based, clustering-based, graph-based, and learning-based techniques. Of these, the graph- and learning-based methods tend to achieve the highest accuracy, but at the highest computational cost.

Graph-based methods are well-suited for image segmentation tasks due to their ability to use contextual information contained in the image, i.e., relationships among pixels and/or regions. The probabilistic graphical model (PGM) known as Markov random fields (MRF) [22] is an example of one such method.

MRFs represent discrete data by modeling neighborhood relationships, thereby consolidating structure representation for image analysis [21].

Despite their high accuracy, MRF optimization algorithms have high computational complexity (NP-hard). Strategies for overcoming the complexity, such as graph-cut techniques, are often restricted to specific types of models (first-order MRFs) [14] and energy functions (regular or submodular) [14]. For higher-order MRFs and non-submodular functions, some strategies using parallelized graph cuts and parallelized Belief Propagation have also been proposed [7,10,12,32]. These approaches, though, typically depend on orderly reduction or submodular functions [34], which are undesirable constraints when dealing with complex and large image datasets because they limit the contextual modeling of the problem.

In order to circumvent such drawbacks, recent works [24,25] have proposed theoretical foundations for distributed parameter estimation in MRF. These approaches make use of a composite likelihood, which enable parallel solutions to subproblems. Under general conditions on the composite likelihood factorizations, the distributed estimators are proven to be consistent. The Linear and Parallel (LAP) [26] algorithm parallelizes naturally over cliques and, for graphs of bounded degree, its complexity is linear in the number of cliques. It is fully parallel and, for log-linear models, it is also data efficient. It requires only the local statistics of the data, i.e., considering only pixel values of local neighborhoods, to estimate parameters.

Perciano *et al.* [30] describe a graph-based model, referred to as Parallel Markov Random Fields (PMRF), which exploits MRFs to segment images. Both the optimization and parameter estimation processes are parallelized using the LAP method, and the implementation is based on C11 multithreading. The first attempt to reimplement the PMRF algorithm is described in [11], where a distributed-memory version of the algorithm is implemented using MPI. Lessley *et al.* [18] reformulates the PMRF algorithm using data parallel primitives implemented in the VTK-m library. This work takes advantage of a new implementation of the maximal cliques problem also using DPPs [19].

Although the previous works study the computational performance of the reimplemented versions of the PMRF algorithm, the correctness of the new versions is emphasized. In the work we present here, we describe a detailed study of shared-memory scalability, as well as collecting hardware performance counters, such as FLOPS/vectorization, memory utilization, instruction counts, and so forth, and use these to perform an in-depth analysis of three shared-memory parallel implementations of the PMRF algorithm: C11-threads, OpenMP, DPP. In the long term, these shared-memory parallel methods would be paired with our distributed-memory parallel implementation [11] to produce a scalable, hybrid-parallel implementation that is portable across HPC platforms and processors.

2.2 Performance and Portability

Open Multi-Processing (OpenMP). OpenMP has been used before to accelerate graph-based algorithms. Recently, Meng *et al.* [24] proposed a parallelization of graph-based machine learning algorithms using OpenMP. The authors also use LAPACK [2] and BLAS [4], which are highly vectorized and multi-threaded using OpenMP, to optimize intensive linear algebra calculations.

Sariyuce *et al.* [31], describe a hybrid implementation of graph coloring using MPI and OpenMP. Ersoy *et al.* [9] proposed a parallel implementation of a shortest path algorithm for time dependent graphs using OpenMP and CUDA. Time dependent shortest path problem (TDSPP) is another example of an NP-hard problem, as the one we tackle in this paper.

We reformulate the PMRF algorithm using OpenMP by targeting loop parallelization over neighborhoods, which is relatively coarse-grained when compared to "inner-loop" parallelization. Load balancing is enabled through OpenMPs dynamic scheduling algorithms.

Data Parallel Primitives (DPP). The primary motivation of using DPPs, particularly those that are amenable to vectorization, is because this approach appears promising for achieving good performance on multi- and many-core architectures. Levesque and Voss, 2017 [20], speculate that vectorized codes may achieve performance gains of as much as 10–30 fold compared to non-vectorized code, with the added benefit of using less power on multi- and many-core architectures. DPPs are amenable to vectorization, and in turn, are capable of high performance on multi- and many-core architectures. This idea is not new, but goes over 20 years to early work by Blelloch [5], who proposed a vector-scan model for parallel computing.

Lessley, et al., 2018 [18] present an implementation of the PMRF algorithm using DPPs. The DPP form of PMRF required a non-trivial reformulation of the reference C++/OpenMP implementation, where reformulation is required to map traditional loop-based computations onto data parallel primitives such as `Sort`, `Scan`, `Reduce`, etc. That work compared performance and scaling differences of DPP-PMRF and C++/OpenMP parallel versions by measuring runtime performance.

The DPP-PMRF implementation relies on the VTK-m library [28], which is a platform-portable framework that provides a set of key DPPs, along with back-end code generation and runtime support for the use of GPUs (CUDA) and multi-core CPUs (TBB [6]) from a single code base [27]. VTK-m achieves parallelization by distributing the work of its DPPs across "threads" using a chunking/blocking model, where a larger collection of work is distributed in chunks or blocks across threads. This basic concept applies to both CPU and GPU implementations.

For the work we present here, we are using the implementation of DPP-PMRF from Lessley et al., 2018 [18], but building upon that previous work in a significant way. Namely, the 2018 study measured only runtime, whereas in the work we present here, we are measuring several different types of hardware

performance counters to gain a better understanding of the factors contributing to absolute runtime performance differences and relative scaling characteristics, and compare those measures with those obtained from traditional OpenMP and threads-parallel implementations of the PMRF algorithm.

3 Design and Implementation

We begin by presenting the baseline, serial MRF-based image segmentation algorithm (Sect. 3.1). The subsequent subsections cover three different parallel implementations: C11-threads (Sect. 3.2), OpenMP (Sect. 3.3), and DPP (Sect. 3.4). Each of these parallel subsections will focus on key parallelization topics, namely work decomposition and the parallel algorithm implementation, with an emphasis on highlighting differences from the baseline implementation.

3.1 The Baseline MRF Algorithm

The baseline MRF algorithm, along with a threads-parallel variant, are described in more detail in Perciano *et al.*, 2016 [30]. The input consists of a grayscale image, an oversegmentation of the input image, and a parameter indicating the desired number of output labels (classes). The oversegmented image is a preliminary segmentation based upon a low-cost computational estimate. For example, a threshold operator can produce an oversegmented image. The oversegmented image is known to be inaccurate, but is inexpensive to compute. It is inaccurate in that it has "too many" segments, or regions, hence the name "oversegmented". The oversegmented image serves as the starting point for MRF optimization, which will merge and change oversegmented regions into a more accurate segmentation.

The pseudocode for the Baseline MRF algorithm is shown in Algorithm 1. It consists of a one-time initialization phase, followed by a compute-intensive, primary parameter estimation optimization phase. The output is a segmented image.

Algorithm 1. Baseline MRF

Require: Original image, oversegmentation, number of output labels
Ensure: Segmented image and estimated parameters
1: Initialize parameters and labels randomly
2: Create graph from oversegmentation
3: Find maximal cliques of the graph
4: Construct k-neighborhoods for all maximal cliques
5: **for** each EM iteration **do**
6:     **for** each neighborhood of the subgraph **do**
7:         Compute MAP estimation
8:     **end for**
9:     Update parameters and labels
10: **end for**

The goal of the initialization phase is the construction of an undirected graph of pixel regions, each with statistically similar grayscale intensities among member pixels. Starting with the original image and the oversegmented version of that image, the algorithm then builds a graph from the oversegmented image, where each vertex V represents a region in the oversegmented image (i.e., a spatially connected region of pixels having similar intensity), and each edge E indicates spatial adjacency of regions.

Next, in the main computational phase, we define an MRF model over the set of vertices, which includes an energy function representing contextual information of the image. In particular, this model specifies a probability distribution over the k-neighborhoods of the graph. Each k-neighborhood consists of the vertices of a maximal clique, along with all neighbor vertices that are within k edges (or hops) from any of the clique vertices; in this study, we use $k = 1$.

The MRF algorithm then performs energy function optimization over each of these neighborhoods. This optimization consists of an iterative invocation of the expectation-maximization (EM) algorithm, which performs parameter estimation using the maximum *a posteriori* (MAP) inference algorithm [13]. The goal of the optimization routine is to converge on the most-likely (minimum-energy) assignment of labels for the vertices in the graph; the mapping of the vertex labels back to pixels yields the output image segmentation.

3.2 The C++/Threads Algorithm

The C++/Threads implementation uses the same input and parameters as the Baseline implementation and performs the same types of computations. The computation is parallelized by dividing the N neighborhoods evenly across each of the T threads. In this case, the first group of N/T neighborhoods is assigned to the first thread, the second N/T to the second thread, and so forth. Processing consists of optimizing a set of neighborhoods using MAP and estimating the parameters for the desired labels (classes). In this implementation, a shared-memory array that hold results from the optimization process is used by all threads on subsequent EM iterations. This shared-memory array is a vector that carries the estimated classes for each vertex of the graph. During the parallel optimization process, the threads are synchronized every time the shared-memory is updated with new estimated values for each vertex.

This threads-based model is coarse-grained parallelism: each thread is responsible for a rather sizeable amount of work, with relatively little interaction between threads. Also, the way the algorithm distributes the work across threads does not take into account the size of the neighborhoods. This can potentially lead to load imbalance, given that the neighborhoods can vary considerably depending on the input and oversegmentation.

Algorithm 2. C++/Threads: Threaded implementation of parallel MRF

Require: Original image, oversegmentation, number of output labels
Ensure: Segmented image and estimated parameters
1: Initialize parameters and labels randomly
2: Create graph from oversegmentation
3: Find maximal cliques of the graph
4: Construct k-neighborhoods for all maximal cliques
5: Partition into T groups of size N/T
6: **for** *In parallel*: each thread processes its N/T group **do**
7:     **for** each EM iteration **do**
8:         **for** each neighborhood of the subgraph **do**
9:             Compute MAP estimation
10:         **end for**
11:         Update parameters and labels
12:     **end for**
13: **end for**

3.3 The C++/OpenMP Algorithm

The OpenMP-parallel version of the MRF algorithm, shown in Algorithm 3, uses the same input and parameters as the Baseline implementation and performs the same types of computations. This version is finer-grained in terms of workload distribution as compared to the C++/Threads version: the inner loop of Algorithm 3 iterates over neighborhoods, of which there are typically many (thousands for 2D images to millions for 3D volumes). We parallelize that neighborhood-iteration loop using OpenMP, and use OpenMP's dynamic thread scheduling algorithms to achieve more even load balance across all threads. The challenge in this problem is that the amount of computation required for each neighborhood varies as a function of the size of the neighborhood and its connectivity to adjacent regions.

Algorithm 3. C++/OpenMP: Parallelization with OpenMP

Require: Original image, oversegmentation, number of output labels
Ensure: Segmented image and estimated parameters
1: Initialize parameters and labels randomly
2: Create graph from oversegmentation
3: Find maximal cliques of the graph
4: Construct k-neighborhoods for all maximal cliques
5: **for** each EM iteration **do**
6:     **for** *OpenMP parallel* each neighborhood of the subgraph **do**
7:         Compute MAP estimation
8:     **end for**
9:     Update parameters and labels
10: **end for**

Compared to the threads implementation, the OpenMP design targets a finer granularity so as to achieve better load balance across threads. One of the objectives of our performance study is to better understand the performance impact of these different design choices.

3.4 VTK-m/DPP

Our VTK-m/DPP implementation [18] is a complete reformulation of the MRF algorithm using data-parallel primitives. To make use of DPPs, the implementation recasts the algorithm into a sequence of DPP-based processing steps, e.g, sequences of operations like `Scan`, `Sort`, and `Reduce`.

Algorithm 4 shows pseudocode for the VTK-m/DPP algorithm. Each of these steps is a complex seqeuence of DPP calls. For example, from [18], the line that reads"Construct k-neighborhoods from maximal cliques in parallel" consists of several DPP operations:

Algorithm 4. VTK-m/DPP: Data parallel primitive version of Markov Random Field algorithm

Require: Original image, oversegmentation, number of output labels
Ensure: Segmented image and estimated parameters
1: *DPP in parallel:* Create graph from oversegmentation
2: *DPP in parallel:* Enumerate maximal cliques of graph
3: Initialize parameters and labels randomly
4: *DPP in parallel:* Construct k-neighborhoods from maximal cliques
5: *DPP in parallel:* Replicate neighborhoods by label
6: **for** each EM iteration **do**
7:     *DPP in parallel:* Gather replicated parameters and labels
8:     **for** each vertex of each neighborhood **do**
9:         *DPP in parallel:* MAP estimation
10:     **end for**
11:     *DPP in parallel:* Update parameters and labels
12: **end for**

1. A `Map` operator finds the count of neighbors that are within 1 edge from the vertex and not a member of the vertex's maximal clique;
2. A `Scan` operator adds the counts of neighbors for the purpose of allocating a neighbors array work buffer;
3. A `Map` operator populates the newly created neighbors array;
4. The `SortByKey` and `Unique` operators remove duplicate neighbors.

There are several significant differences between the VTK-m/DPP, C++/-OpenMP, and C++/Threads implementations of this method. One is the level of granularity in parallelization. The C++/Threads is the coarsest decomposition, C++/OpenMP in the middle, and the VTK-m/DPP is the finest level of granularity of parallelization. Each of the DPPs is distributed in chunking fashion

across each of the T available execution threads. Another key difference is in the relative level of "verboseness" of the algorithm itself. While we did not count the number of lines of code, as we shall see in the Results section (Sect. 4.3), there is a significant difference in the number of instructions executed by each of these implementations.

4 Experiment and Results

The experiments in this section serve to answer two primary questions. First, we are interested in how well the different implementations perform on a single-socket study: what are the key performance characteristics of each version? Second, we collect hardware performance counters to understand how well each implementation vectorizes and makes use of the memory hierarchy: what are the factors that lead to these performance characteristics? Sect. 4.1 describes the source datasets and the computational platform that we use for the experiments. Sect. 4.2 presents results of the study, which we discuss in Sect. 4.3.

4.1 Datasets, Computational Platforms, and Software

Datasets. We are using an experimental dataset that was generated by the Lawrence Berkeley National Laboratory Advanced Light Source X-ray beamline 8.3.2[2] [8] for all tests. This dataset contains cross-sections of a geological sample and conveys information regarding the x-ray attenuation and density of the scanned material as a gray scale value. The scanned samples are pre-processed using a separate software that provides reconstruction of the parallel beam projection data into a 1 GB stack of 500 image slices with dimensions of 1290×1305.

For our experiments, we use two augmented versions of this dataset, where we replicate the data of each cross-section by mirroring in both the X and Y dimensions of the data, resulting in one 3.3 GB stack of 500 image slices with dimensions of 2580×2610 (referred to as the 'Sandstone2K' dataset), and one 6.6 GB stack of 500 image slices with dimensions of 5160×5220 (referred to as the 'Sandstone5k' dataset).

Hardware Platforms
Intel Xeon Phi. `Cori.nersc.gov` is a Cray XC40 system comprised of 2,388 nodes containing two 2.3 Ghz 16-core Intel Haswell processors and 128 GB DDR4 2133 MHz memory, and 9,688 nodes containing a single 68-core 1.4 GHz Intel Xeon Phi 7250 (Knights Landing) processor and 96 GB DDR4 2400 GHz memory. For our experiments, we use the KNL processor.[3] Compiler: Intel ICC 19.0.3.199.

[2] http://microct.lbl.gov.
[3] Cori configuration page: http://www.nersc.gov/users/computational-systems/cori/configuration/.

Ivy Bridge. `Allen.lbl.gov` is an Intel(R) Xeon(R) CPU E5-2609 v2 containing two 2.5 GHz 4-core Intel Xeon Ivy Bridge EN/EP/EX processors and 32 GB of memory. Compiler: Intel ICC 19.1.0.166.

Software Environment. The software environment in these tests consists of several different codes, each of which we describe below.

Oversegmentation. We use a custom implementation of the Simple Linear Iterative Clustering (SLIC) method [1]. This implementation will take as input 2D images or 3D volumes of scalar values (gray-level images), and output 2D images or 3D volumes at the same resolution as the input data, but where output pixels/voxels are region label values, rather than pixel/voxel luminosity. We prepare using the input images/volumes described above, and process them using the following SLIC parameters: superpixel size = 80, compactness = 10.

C++/OpenMP and C++/Threads. The C++/Threads version of PMRF is implemented using C11 multithreading for parallelization. The C++/OpenMP algorithm is implemented with OpenMP 4.5. We take advantage of OpenMP loop parallelism constructs to achieve outer-parallelism over MRF neighborhoods, and make use of OpenMP's dynamic scheduling algorithm in the performance studies.

VTK-m/DPP. The VTK-m/DPP algorithm is implemented using the platform-portable VTK-m toolkit [27], and coded to VTK-m API version 1.3.0. In our experiments, we configured VTK-m for parallelism on the CPU by enabling an OpenMP backend, and set the VTK-m index integer (vtkm::Id) size to 64 bits.

Hardware Performance Counters. For measuring hardware performance counters on CPU platforms, we made use of `likwid-perfctr`, which is part of the LIKWID toolsuite [33]. LIKWID is a collection of command line programs that facilitate performance-oriented program development and production in x86 multicore environments under Linux. Using LIKWID 4.3.4 on Allen/Ivy Bridge and 4.3.0 on Cori/KNL, we collected and analyzed several different performance counters and restricted these measures to the PGM graph optimization phase only by using LIKWID's marker API:

- Counts of total number of double-precision scalar and vector instructions executed (`FLOPS_DP`), as well as total number of all scalar and vector instructions executed (`UOPS_RETIRED_*`).
- Measures related to L2 cache: L2 request rate, miss rate, and miss ratio (`L2CACHE`).
- Vectorization ratio. On Ivy Bridge, LIKWID reports this directly. On KNL, we compute this ratio to be $V/(V+S)$, where V is the count of vector ("packed") operations, and S is the count of scalar operations.

4.2 Performance and Scalability Studies: Parallel MRF

Here, we present the results of performance and scaling studies of the three different PMRF implementations (VTK-m/DPP, C++/OpenMP, C++/Threads) on two different platforms (KNL, Ivy Bridge). The primary objective is to compare their runtime performance and scalability, and to examine hardware counters to gain deeper insight into the performance characteristics of each method. The discussion and analysis of these results appears in Sect. 4.3, which follows.

Runtime. The first performance study question we examine is a comparison of runtimes among the implementations used. We executed all the codes at varying levels of concurrency on the KNL platform and Ivy Bridge platforms using two different datasets (sandstone2k and sandstone5k). The speedup plots for the datasets on both platforms are shown in Fig. 2 and Fig. 3.

Speedup is defined as $S(n,p) = \frac{T^*(n)}{T(n,p)}$ where $T(n,p)$ is the time it takes to run the parallel algorithm on p processes with an input size of n, and $T^*(n)$ is the time for the best serial algorithm on the same input.

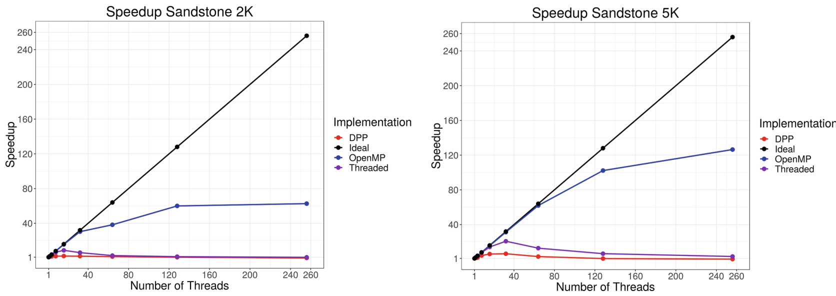


Fig. 2. Speedup of the Sandstone2K and Sandstone5K datasets on Cori. The horizontal axis is the concurrency level and the vertical axis measures the speedup.

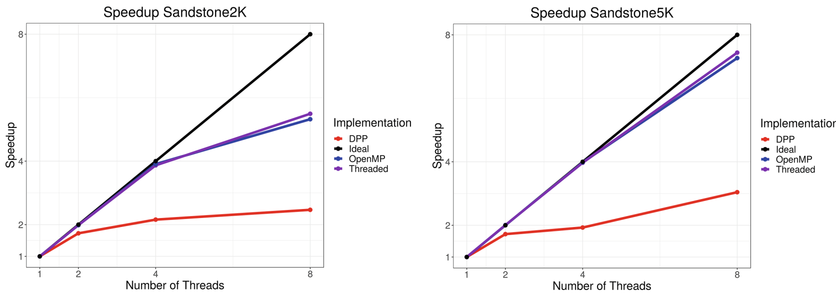


Fig. 3. Speedup of the Sandstone2K and Sandstone5K datasets on the Ivy Bridge platform. The horizontal axis is the concurrency level and the vertical axis measures the speedup.

Examining these two speedup plots, we notice that although the VTK-m/DPP version presents much faster runtimes at lower concurrencies, this implementation shows the worst speedups for both platforms. On the Ivy Bridge platform, both the C++/Threads and C++/OpenMP versions present the best speedup values with very similar results. On the other hand, on the KNL platform, the C++/Threads version presents a similar speedup compared to the VTK-m/DPP version.

Hardware Performance Counters. To gain a deeper understanding into code performance, we collect hardware performance counters using LIKWID on the KNL and Ivy Bridge platforms for the three different implementations. Table 1 shows the results of the three different implementations run on the KNL platform. Here, we vary concurrency across the range of 1, 2, ..., 256. Using LIKWID, we record the counts for the `FLOPS_DP` and `L2CACHE` performance counter groups. Results for the Ivy Bridge platform tests are shown in Table 2, where concurrency varies across 1, 2, ..., 8.

Table 1. KNL Platform and Hardware Performance Counters for the Sandstone5K Dataset. Legend for counters: FLOPS: FLOPS_DP ($*10^9$); Vector%: Vectorization Ratio (Proxy); L2 Miss Ratio: average % across all threads at a given concurrency.

Counter	Code ver.	Concurrency								
		1	2	4	8	16	32	64	128	256
Runtime (secs)	VTK-m/DPP	5.78	3.93	3.01	1.33	0.94	0.90	1.84	6.65	27.39
	C++/OpenMP	143.56	72.75	36.48	18.25	9.14	4.58	2.31	1.40	1.13
	C++/Threads	140.16	70.24	35.48	18.09	9.89	6.73	10.92	21.60	43.23
		1	2	4	8	16	32	64	128	256
FLOPS	VTK-m/DPP	0.85	0.86	0.86	0.86	0.86	0.86	0.86	0.88	0.88
	C++/OpenMP	49.32	49.32	49.32	49.32	49.32	49.32	49.32	49.32	49.32
	C++/Threads	45.39	45.49	45.59	45.79	46.19	47.00	48.62	51.84	57.66
		1	2	4	8	16	32	64	128	256
L2 Miss Ratio %	VTK-m/DPP	0.01	0.20	0.39	0.97	2.86	7.72	24.79	61.05	64.66
	C++/OpenMP	0.01	0.01	0.02	0.09	0.09	0.05	0.11	1.22	8.12
	C++/Threads	0.01	0.01	0.06	0.16	0.28	0.36	0.42	0.94	1.57
		1								
Vector %	VTK-m/DPP	43.48%								
	C++/OpenMP	51.44%								
	C++/Threads	46.89%								

4.3 Discussion and Analysis

The VTK-m/DPP code is executing far fewer floating point instructions than its C++/OpenMP and C++/Threads counterparts. For all the test results we present, the runtime difference between the DPP (VTK-m/DPP) and non-DPP

(C++/OpenMP, C++/Threads) versions appears to be proportional to the difference in the amount of floating point instructions being executed. While the DPP code is solving the same set of numerical equations as the non-DPP code (for the MRF optimization), it does so using a completely different algorithmic formulation. The DPP code design involved a significant refactorization of the MRF optimization algorithm to map it to data parallel primitives [18]. This reordering has resulted in significantly fewer operations being required to perform the computation, and is one of the primary findings of this study.

Vectorization Ratios. In both Table 1 and Table 2, we report a Vectorization Ratio only for the serial configuration because this value does not change in a significant way with increasing concurrency: the algorithm's complexity is primarily dependent upon problem size.

On the KNL platform, we see vectorization ratios that are comparable across all three implementations, in the range of about 43%–51%. This result suggests that the looping structures in all three implementations are amenable to a reasonable level of automatic vectorization by the compiler on the KNL platform.

On the Ivy Bridge platform, the C++/OpenMP and C++/Threads versions show vectorization ratios above 70%, while the VTK-m/DPP version shows a much lower vectorization ratio of about 18%. There are two likely factors

Table 2. Ivy Bridge Platform and Hardware Performance Counters for the Sandstone5K Dataset. Legend for counters: FLOPS: (Double Precision Scalar FLOPS + Double Precision Vector FLOPS) / (10^9); Vector%: Vectorization Ratio; L2 Miss Ratio: average % across all threads at a given concurrency.

Counter/Measure	Code version	Concurrency			
Runtime (secs)		1	2	4	8
	VTK-m/DPP	2.51	1.46	1.30	0.83
	C++/OpenMP	13.34	6.66	3.35	1.83
	C++/Threads	13.94	7.00	3.51	2.16
FLOPS ($*10^9$)		1	2	4	8
	VTK-m/DPP	0.47	0.33	0.33	0.33
	C++/OpenMP	7.14	7.13	7.13	7.13
	C++/Threads	7.25	7.26	7.26	7.27
L2 Miss Ratio %		1	2	4	8
	VTK-m/DPP	0.26	0.26	0.25	0.25
	C++/OpenMP	0.05	0.07	0.05	0.05
	C++/Threads	0.04	0.06	0.06	0.06
Vector %		1			
	VTK-m/DPP	18.16%			
	C++/OpenMP	73.31%			
	C++/Threads	70.43%			

contributing to this difference. The first is in the code itself: the C++/OpenMP and C++/Threads codes implement their computations using C++ vector objects, and these are likely easier for the compiler to auto-vectorize compared to code that performs explicit blocking and chunking, as is the case with VTK-m internals, which will take a large DPP operation and decompose it into smaller chunks, which are then executed in parallel by one of several backends (TBB, OpenMP, or CUDA). With explicit blocking and chunking, there may be an adverse interplay between how VTK-m blocks and chunks and what the compiler needs for a given architecture. This particular issue merits further study to better understand this interplay.

The second reason concerns variation in how the compiler auto-vectorizes code for each architecture. What works well on one architecture may not work so well on a different architecture: we see this with the VTK-m/DPP code base where the Intel compiler auto-vectorizes code to produce 43.48% vectorization on the KNL platform, but is able to manage only 18.16% on the Ivy Bridge platform. With the hardware performance counters we have access to with LIKWID, we are unable to discern precisely which type of vector instructions are being executed (e.g., SSE, AVX, AVX512) on each platform, which would in term provide more useful insights.

At the outset of this study, we had the expectation that the VTK-m/DPP code would have significantly better vectorization characteristics, which would then account for its significantly faster runtime, particularly as we observed in earlier studies [18]. Instead, what we see are comparable levels of vectorization on the KNL (43%–51%), and a lower vectorization level on Ivy Bridge (18%).

It turns out that there are other factors that, in this study, have much more impact on code performance than vectorization, namely the absolute number of instructions executed. One of the primary findings of this study is that our refactoring a complex graph algorithm (PMRF) to use DPPs results in significantly fewer instructions being executed compared to implementations using C++/OpenMP and C++/Threads.

Scalability. These studies show differing levels of scalability, as evidenced in the speedup charts shown in Fig. 2. The VTK-m/DPP code on the KNL platform shows decreasing runtime up to about 32 cores, after which it increases in runtime. Looking at the performance counters in Table 1, we see a corresponding increase in the L2 Cache Miss ratio. The L2 cache misses are due to how KNL shares L2 cache across hardware threads, where increasing the number of threads to exceed the number of cores causes the amount of L2 cache available to each core to be reduced. In other words, if there is one thread per core, it will use all of the L2 cache, if two threads share a core, each thread has one-half of the L2 cache, and if three or four threads share a core, then each thread has access to 1/4 of the L2 cache.

On the KNL, the C++/Threads implementation shows decreasing runtime up to about 32 cores, after which point the runtime increases significantly. Whereas the VTK-m/DPP code shows significant L2 cache misses at higher concurrency, the C++/Threads version does not. Instead, this performance

difference between C++/Threads and C++/OpenMP at higher concurrency is most likely the result of a highly optimized OpenMP loop parallelization that is provided by the compiler, an effect that does not become readily apparent until higher degrees of node-level concurrency on the KNL platform.

On the Ivy Bridge platform, all implementations exhibit better scalability than on the KNL platform. This is most likely the result of a large L3 cache that is shared across all cores, something that is not present on the KNL platform. However, the Ivy Bridge study only goes up to 8 threads, and the declined in speedup shown in the KNL study (Fig. 2) does not begin until higher levels of concurrency. Therefore, while we may see a decline in speedup on the Ivy Bridge at higher concurrency, since that platform has only 8 cores, our study only goes up to 8-way concurrency.

Platform Portability Issues. One of the objectives of OpenMP and VTK-m is to provide platform portability, so that a given code implementation can be run, without modification, on CPU and GPU platforms. We have demonstrated in previous work [18] that the VTK-m/DPP is capable of running on the GPU platform. For the C++/OpenMP implementation, there are significant restrictions and limitations on OpenMP in terms of what kind of code can be processed successfully to emit device code. At the present time, our C++/OpenMP implementation would require significant changes, including, but not limited to, eliminating the use of "ragged arrays", which are not naturally supported by OpenMP on the GPU. This will be the subject of future work. Meanwhile, the KNL platform in these studies allows us to go to 256-way parallel for the purposes of performance analysis. This degree of node-level concurrency is expected to be commonplace on future platforms.

5 Conclusion and Future Work

One of the objectives of this work has been to understand the performance characteristics of three different approaches for doing shared-memory parallelization of a probabilistic graphical modeling optimization code, which serves as the basis for a highly accurate, and scalable method for scientific image segmentation. The work is motivated by the need to improve throughput of scientific analysis tools in light of increasing sensor and detector resolution. The three parallelization methods consist of two that are "traditional" (C++/OpenMP and C++/Threads) and one that is "non-traditional" (VTK-m/DPP).

At the outset of this work, we expected that the VTK-m/DPP implementation was running faster than the other two due to better vectorization characteristics. The results of our performance study point to a different reason for the performance difference: the VTK-m/DPP version executes many fewer instructions. The reason is because the process of reformulating a complex, graphical model optimization code to use sequences of DPPs results in runtime code that is more terse and efficient in terms of number of computations needed to produce the same answer as the corresponding C++/Threads and C++/OpenMP

formulations. This bit of insight, and the performance analysis methodology we used, is the primary contribution of this paper. To our knowledge, this study is the first of its kind: an in-depth performance analysis of codes based on DPPs, OpenMP, and threads.

Future work will include pressing deeper into the topic of platform portability. While our VTK-m/DPP implementation can run on both CPU and GPU platforms, owing to the capabilities of the underlying DPP implementation, which is based on VTK-m, our OpenMP codes are not yet capable of running on GPU platforms. For OpenMP to emit code that runs on a GPU, the application must conform to a strict set of memory access patterns. Future work will include redesigning our code so that it does conform to those limitations.

The topic of platform portability and performance is of significant concern as computational platforms increase in concurrency, particularly at the node level. For that reason, this particular study is timely, for it sheds light on the performance characteristics of a non-trivial, data-intensive code implemented with three different methodologies, one of which is relatively new and holds promise.

Acknowledgment. This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, through the grant "Scalable Data-Computing Convergence and Scientific Knowledge Discovery," program manager Dr. Laura Biven, and the Center for Applied Mathematics for Energy Related Applications (CAMERA). We also thank the LBNL ALS division for the data and NERSC for the computational resources.

References

1. Achanta, R., Shaji, A., Smith, K., Lucchi, A., Fua, P., Süsstrunk, S.: Slic superpixels compared to state-of-the-art superpixel methods. IEEE Trans. Pattern Anal. Mach. Intell. **34**(11), 2274–2282 (2012). https://doi.org/10.1109/TPAMI.2012.120. https://ieeexplore.ieee.org/document/6205760
2. Anderson, E., et al.: Lapack: a portable linear algebra library for high-performance computers. In: Proceedings of the 1990 ACM/IEEE Conference on Supercomputing, pp. 2–11. Supercomputing 1990, IEEE Computer Society Press, Los Alamitos, CA, USA (1990)
3. Bethel, E.W., Greenwald, M., van Dam, K.K., Parashar, M., Wild, S.M., Wiley, H.S.: Management, analysis, and visualization of experimental and observational data - the convergence of data and computing. In: Proceedings of the 2016 IEEE 12th International Conference on eScience. Baltimore, MD, USA, October 2016
4. Blackford, L.S., et al.: An updated set of basic linear algebra subprograms (BLAS). ACM Trans. Math. Softw. **28**(2), 135–151 (2002). https://doi.org/10.1145/567806.567807
5. Blelloch, G.E.: Vector Models for Data-parallel Computing. MIT Press, Cambridge (1990)
6. Corporation, I.: Introducing the Intel Threading Building Blocks, May 2017. https://software.intel.com/en-us/node/506042

7. Delong, A., Boykov, Y.: A scalable graph-cut algorithm for n-d grids. In: IEEE Conference on Computer Vision and Pattern Recognition (2008)
8. Donatelli, J., et al.: Camera: the center for advanced mathematics for energy research applications. Synchrotron Radiation News **28**(2), 4–9 (2015)
9. Ersoy, M.A., Özturan, C.: Parallelizing shortest path algorithm for time dependent graphs with flow speed model. In: 2016 IEEE 10th International Conference on Application of Information and Communication Technologies (AICT), pp. 1–7, October 2016. https://doi.org/10.1109/ICAICT.2016.7991833
10. Eslami, H., Kasampalis, T., Kotsifakou, M.: A GPU implementation of tiled belief propagation on markov random fields. In: 2013 Eleventh ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2013), pp. 143–146 (Oct 2013)
11. Heinemann, C., Perciano, T., Ushizima, D., Bethel, E.W.: Distributed memory parallel markov random fields using graph partitioning. In: Fourth International Workshop on High Performance Big Graph Data Management, Analysis, and Mining (BigGraphs 2017), in conjunction with IEEE BigData 2017, December 2017
12. Jamriska, O., Sykora, D., Hornung, A.: A cache-efficient graph cuts on structured grids. In: IEEE Conference on Computer Vision and Pattern Recognition, pp. 3673–3680 (2012)
13. Koller, D., Friedman, N.: Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning. MIT Press, Cambridge (2009)
14. Kolmogorov, V., Zabin, R.: What energy functions can be minimized via graph cuts? IEEE Trans. Pattern Anal. Mach. Intell. **26**(2), 147–159 (2004)
15. Larsen, M., Labasan, S., Navrátil, P., Meredith, J., Childs, H.: Volume rendering via data-parallel primitives. In: Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV), pp. 53–62. Cagliari, Italy, May 2015
16. Larsen, M., Meredith, J., Navrátil, P., Childs, H.: Ray-tracing within a data parallel framework. In: Proceedings of the IEEE Pacific Visualization Symposium, pp. 279–286. Hangzhou, China, April 2015
17. Lessley, B., Moreland, K., Larsen, M., Childs, H.: Techniques for data-parallel searching for duplicate elements. In: Proceedings of IEEE Symposium on Large Data Analysis and Visualization (LDAV), pp. 1–5. Phoenix, AZ, October 2017
18. Lessley, B., Perciano, T., Heinemann, C., Camp, D., Childs, H., Bethel, E.W.: DPP-PMRF: rethinking optimization for a probabilistic graphical model using data-parallel primitives. In: 8th IEEE Symposium on Large Data Analysis and Visualization (LDAV). Berlin, Germany, October 2018
19. Lessley, B., Perciano, T., Mathai, M., Childs, H., Bethel, E.W.: Maximal clique enumeration with data-parallel primitives. In: IEEE Large Data Analysis and Visualization. Phoenix, AZ, USA, October 2017
20. Levesque, J., Vose, A.: Programming for Hybrid Multi/Many-core MPP Systems. Chapman & Hall, CRC Computational Science, CRC Press/Francis&Taylor Group, Boca Raton, November 2017, preprint
21. Lezoray, O., Grady, L.: Image Processing and Analysis with Graphs: Theory and Practice. CRC Press, Boca Raton (2012)
22. Li, S.Z.: Markov Random Field Modeling in Image Analysis (2013). https://doi.org/10.1007/978-1-84800-279-1
23. Li, S., Marsaglia, N., Chen, V., Sewell, C., Clyne, J., Childs, H.: Achieving portable performance for wavelet compression using data parallel primitives. In: Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV), pp. 73–81. Barcelona, Spain, June 2017

24. Meng, Z., Wei, D., Wiesel, A., Hero, A.O.: Distributed learning of gaussian graphical models via marginal likelihoods. In: The Sixteenth International Conference on Artificial Intelligence and Statistics, pp. 39–47 (2013)
25. Meng, Z., Wei, D., Wiesel, A., Hero, A.O.: Marginal likelihoods for distributed parameter estimation of gaussian graphical models. IEEE Trans. Signal Process. **62**(20), 5425–5438 (2014)
26. Mizrahi, Y.D., Denil, M., de Freitas, N.: Linear and parallel learning of markov random fields. Proc. Int. Conf. Mach. Learn. **32**, 1–10 (2014)
27. Moreland, K.: VTK-m website, May 2017. http://m.vtk.org
28. Moreland, K., et al.: VTK-m: accelerating the visualization toolkit for massively threaded architectures. IEEE Comput. Graph. Appl. (CG&A) **36**(3), 48–58 (2016)
29. Perciano, T., et al.: Insight into 3D micro-CT data: exploring segmentation algorithms through performance metrics. J. Synchrotron Radiat. **24**(5), 1065–1077 (2017)
30. Perciano, T., Ushizima, D.M., Bethel, E.W., Mizrahi, Y.D., Parkinson, D., Sethian, J.A.: Reduced-complexity image segmentation under parallel markov random field formulation using graph partitioning. In: 2016 IEEE International Conference on Image Processing (ICIP). pp. 1259–1263, September 2016
31. Sariyuce, A.E., Saule, E., Catalyurek, U.V.: Scalable hybrid implementation of graph coloring using MPI and OPENMP. In: Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & Ph.D. Forum, pp. 1744–1753. IPDPSW 2012, IEEE Computer Society, Washington, DC, USA (2012). https://doi.org/10.1109/IPDPSW.2012.216
32. Shekbovstov, A., Hlavac, V.: A distributed mincut/maxflow algorithm combining augmentation and push-relabel. In: International Journal of Computer Visualization (2012)
33. Treibig, J., Hager, G., Wellein, G.: Likwid: a lightweight performance-oriented tool suite for x86 multicore environments. In: Proceedings of the 2010 39th International Conference on Parallel Processing Workshops, pp. 207–216. ICPPW 2010, IEEE Computer Society, Washington, DC, USA (2010). https://doi.org/10.1109/ICPPW.2010.38
34. Wang, C., Komodakis, N., Paragios, N.: Markov random field modeling, inference, learning in computer vision and image understanding: a survey. Comput. Vis. Image Understand. **117**(11), 1610–1627 (2013)