Title

OpenMP Parallelization and Optimization of Graph-Based Machine Learning Algorithms

Permalink

https://escholarship.org/uc/item/7hk0j5kk

ISBN

978-3-319-45549-5

Authors

Meng, Zhaoyi
Koniges, Alice
He, Yun Helen
et al.

Publication Date

2016

DOI

10.1007/978-3-319-45550-1_2

Peer reviewed

OpenMP Parallelization and Optimization of Graph-based Machine Learning Algorithms

Zhaoyi Meng, Alice Koniges, Yun (Helen) He, Samuel Williams,
Thorsten Kurth, Brandon Cook, Jack Deslippe, and Andrea L. Bertozzi

University of California, Los Angeles, US
Lawrence Berkeley National Laboratory, US
mzhy@ucla.edu
aekoniges@lbl.gov

Abstract. We investigate the OpenMP parallelization and optimization of two novel data classification algorithms. The new algorithms are based on graph and PDE solution techniques and provide significant accuracy and performance advantages over traditional data classification algorithms in serial mode. The methods leverage the Nystrom extension to calculate eigenvalue/eigenvectors of the graph Laplacian and this is a self-contained module that can be used in conjunction with other graph-Laplacian based methods such as spectral clustering. We use performance tools to collect the hotspots and memory access of the serial codes and use OpenMP as the parallelization language to parallelize the most time-consuming parts. Where possible, we also use library routines. We then optimize the OpenMP implementations and detail the performance on traditional supercomputer nodes (in our case a Cray XC30), and predict behavior on emerging testbed systems based on Intel's Knights Corner and Landing processors. We show both performance improvement and strong scaling behavior. A large number of optimization techniques and analyses are necessary before the algorithm reaches almost ideal scaling.

Keywords: semi-supervised, unsupervised, data, algorithms, OpenMP, optimization

1 Introduction

We detail the OpenMP parallelization of two new data classification algorithms. A classification algorithm sorts the data into different classes such that the similarity within a class is stronger than that between different classes. This is a standard problem in machine learning. Recently, novel algorithms have been proposed [1] that are motivated by PDE-based image segmentation methods and are modified to apply to discrete data sets [4]. Serial results show that these algorithms improve both accuracy of solution and efficiency of the computation and can be potentially faster in parallel than various classification algorithms such as spectral clustering with k-means [6]. In this paper we describe parallel implementations and optimizations of the new algorithms. We focus on shared memory many-core parallelization schemes that will be applicable to next generation architectures such as the upcoming Intel Knights Landing processor. After analyzing the computational hotspots, we find that an optimized implementation of the Nyström eigensolver is the computational challenge. We implement directive-based OpenMP parallelization on the most time-consuming part and implement steps of optimizations to speed up and achieve almost ideal performance.

The rest of this paper is organized as follows: Section 2 presents the background of the image classification algorithms and the Nyström extension eigensolver. In Section 3 we discuss Math library usage and optimization for the serial code. We show our OpenMP parallelization strategies and optimization steps in Section 4. Finally, Section 5 presents some conclusions and future work.

2 Graph-based Classification Algorithms

2.1 Introduction

We approach the classification problem using graph cut ideas. The novel classification algorithms consider each data point as a node in a weighted graph and the similarity (weight) between two nodes Z_i and Z_j is given by formula:

$$w_{ij} = exp(-dis(Z_i, Z_j)/\tau), \tag{1}$$

where τ is a parameter [5, 6]. The weight matrix is $W = \{w_{ij}\}$. In this paper, we use cosine distance since we use the hyperspectral imagery as the test data set and cosine distance is standard in this field. So

$$dis(Z_i, Z_j) = \frac{< Z_i, Z_j >}{||Z_i||_2||Z_j||_2}. \tag{2}$$

The classification problem is approached using ideas from graph-cuts [2]. Given a weighted undirected graph, the goal is to find the minimum cut (measured by a summation of the weights along the graph cut) for this problem. This is equivalent to assigning a scalar or vector value u_i to each i^{th} data point and minimizing the graph total variation (TV) $\sum_{ij} |u_i - u_j| w_{ij}$ [3]. Instead of directly solving a graph-TV minimization problem, we transform the graph TV to graph-based Ginzburg-Laudau (GL) functional [8]:

$$E(u) = \epsilon < L_s u, u > + \frac{1}{\epsilon} \sum_i (W(u_i)) \tag{3}$$

where $W(u)$ is a double well potential, for example $W(u) = \frac{1}{4}(u^2-1)^2$ in a binary partitioning and multi-well potential in k dimensions (same as the number of classes). L_s is the normalized symmetric graph Laplacian which is defined as $L = I - D^{-\frac{1}{2}} W D^{-\frac{1}{2}}$, where D is a diagonal matrix with diagonal elements $d_i = \sum_{j \in V} w(i, j)$.

In the vanishing ϵ limit we recover the graph TV functional [7]. Different fidelity items are added to GL functional for semi-supervised and unsupervised learning respectively. The GL functional is minimized using the MBO scheme [10], in which one alternates solving the heat (diffusion) equation for u and thresholding to maintain distinct class structure. Computation of the entire graph Laplacian is prohibitive for large data so we use the Nyström extension to randomly sample the graph and compute a modest number of leading eigenvalues and eigenfunctions of the graph Laplacian [9]. By projecting all vectors onto this sub-eigenspace, the iteration step reduces to a simple coefficient update.

2.2 Semi-supervised and Unsupervised Algorithms

We outline the semi-supervised and the unsupervised algorithms. For the semi-supervised algorithm, the fidelity (a small amount of "ground truth") is known and the rest needs to be classified according to the categories of the fidelity. For the unsupervised algorithm, there is no prior knowledge of the labels of the data. We use the Nyström extension algorithm beforehand for both algorithms to calculate the eigenvalues and eigenvectors as the inputs. In practice, these two algorithms converge very fast and give accurate classification results.

Semi-supervised Graph MBO Algorithm [10]

```
1. Input eigenvectors matrix $\Phi$, eigenvalues $\{\lambda_k\}_{k=1}^M$ and fidelity.
2. Initialize $u^0$, $d^0 = \mathbf{0}$, $a^0 = \Phi^T \cdot u_0$.
3. While $\frac{||u^{n+1}-u^n||_2^2}{||u^{n+1}||_2^2} < \alpha = 0.0000001$ do
    a. Heat equation
        1). $a_k^{n+1} = a_k^n \cdot (1 - dt \cdot \lambda_k) - dt \cdot d_k^n$
        2). $y = \Phi \cdot a^{n+1}$
        3). $d^{n+1} = \Phi^T \cdot \mu(y - u^0)$,
    b. Thresholding
        $u_i^{n+1} = e_r, r = \arg\max_j y_i$
    c. Updating $a$
        $a^{n+1} = \Phi^T \cdot u^{n+1}$
```

Unsupervised Graph MBO Algorithm [11]

```
1. Input data matrix $f$, eigenvector matrix $\Phi$, eigenvalues $\{\lambda_k\}_{k=1}^N$.
2. Initialize $u^0$, $a^0 = \Phi^T \cdot u^0$
3. While $\frac{||u^{n+1}-u^n||_2^2}{||u^{n+1}||_2^2} < \alpha = 0.0000001$ do
   a. Updating $c$
      $c_k^{n+1} = \frac{<f,u_k^{n+1}>}{\sum_{i=1}^N u_{ki}}$
   b. Heat equation
        1. $a_k^{n+\frac{1}{2}} = a_k^n \cdot (1 - dt \cdot \lambda_k)$
        2. Calculating matrix $P$, where $P_{i,j} = ||f_i - c_j||_2^2$
        3. $y = \Phi \cdot a_k^{n+\frac{1}{2}} - dt \cdot \mu P$
```

```
c. Thresholding
    $u_i^{n+1} = e_r, r = \arg\max_j y_i$
d. Updating $a$
    $a^{n+1} = \Phi^T \cdot u^{n+1}$
```

2.3 Nyström Extension Method

In both semi-supervised and unsupervised algorithms, we calculate the leading eigenvalues and eigenvectors of the graph Laplacian using Nyström method [9] to accelerate the computation. This is achieved by calculating an eigendecomposition on a smaller system of size $M << N$ and then expanding the results back up to N dimensions. The computational complexity is almost $O(N)$. We can set $M << N$ without any significant decrease in the accuracy of the solution.

Suppose $Z = \{Z_k\}_{k=1}^N$ is the whole set of nodes on the graph. By randomly selecting a small subset X, we can partition Z as $Z = X \bigcup Y$, where X and Y are two disjoint set, $X = \{Z_i\}_{i=1}^M$ and $Y = \{Z_j\}_{j=1}^{N-M}$ and $M << N$. The weight matrix W can be written as

$$W = \begin{bmatrix} W_{XX} \; W_{XY} \\ W_{YX} \; W_{YY} \end{bmatrix},$$

where W_{XX} denotes the weights of nodes in set X, W_{XY} denotes the weights between set X and set Y, $W_{YX} = W_{XY}^T$ and W_{YY} denotes the weights of nodes in set Y. It can be shown that the large matrix W_{YY} can be approximated by $W_{YY} \approx W_{YX} W_{XX}^{-1} W_{XY}$, and the error is determined by how many of the rows of W_{XY} span the rows of W_{YY}. We only need to compute W_{XX}, $W_{XY} = W_{YX}^T$, and it requires only $(|X| \cdot (|X| + |Y|))$ computations versus $(|X| + |Y|)^2$ when the whole matrix is used. For the data set we use in this paper, $M = 100$ and $N = 13,475,840$.

Nyström Extension Algorithm

```
1. Input a set of features $Z = \{Z_i\}_{i=1}^N$.
2. Partition the set $Z$ into $Z = X \cup Y$, where $X$ consists of $M$ randomly selected elements.
3. Calculate $W_{XX}$ and $W_{XY}$ using formula (1).
4. Calculate $d_X = W_{XX} 1_L + W_{XY} 1_{N-L}$ and $d_Y = W_{YX} 1_L + (W_{YX} W_{XX}^{-1} W_{XY}) 1_{N-L}$.
5. Calculate $s_X = \sqrt{d_X}$ and $s_Y = \sqrt{d_Y}$.
6. Calculate $W_{XX} = W_{XX}./(s_X s_X^T)$ and $W_{XY} = W_{XY}./(s_X s_Y^T)$.
7. Calculate eigendecomposition $W_{XX} = B_X \Gamma B_X^T$ (using the SVD).
8. Calculate $S = B_X \Gamma^{-1/2} B_X^T$ and $Q = W_{XX} + S(W_{XY} W_{YX})S$.
9. Calculate eigendecomposition $Q = A \Theta A^T$ (using the SVD).
10. Form eigenvector matrix $\Phi = \begin{bmatrix} B_X \Gamma^{1/2} \\ W_{YX} B_X \Gamma^{-1/2} \end{bmatrix} B_X^T (A \Theta^{-1/2})$.
11. Output $\Phi$ and $\{\lambda_i\}_{i=i}^N$,where $\lambda_k = 1 - \theta_k$ with $\theta_k$ the $k$th diagonal element of $\Theta$.
```

3 Math Library Usage and Optimizations

All the data are in matrix form and there are intensive linear algebra calculations. Also, we apply Singular Value Decomposition (SVD) to two small matrices. So, we make use of the LAPACK (Linear Algebra PACKage) and BLAS (Basic Linear Algebra Subprograms) libraries in the codes. The LAPACK provides routines for the SVD and the BLAS provides routines for vector-vector (Level 1), matrix-vector (Level 2) and matrix-matrix (Level 3) operations. BLAS and LAPACK are also highly vectorized and multithreaded using OpenMP.

We use the Intel Performance Tool VTune Amplifier to analyze the performance and find bottlenecks [18]. The hotspots collection shows some computationally expensive parts are related to calculating the inner product of two vectors. In the unsupervised graph MBO algorithm, this operation occurs when calculating the matrix P and takes 84% of the run time. Also, it occurs when calculating the matrix W_{XY} in the Nyström extension algorithm and takes 90% of the run time. We optimize this procedure by forming all the vectors into matrices and doing the inner product of two matrices. In this way, we make use of BLAS3 (matrix-matrix) instead of BLAS1 (vector-vector). The part of calculating the matrix P in the unsupervised algorithm is 22.5× faster using BLAS3. This optimization is based on the fact that BLAS1,2 are memory bound and BLAS 3 is computational bound [12].

4 Parallelization of the Nyström Extension

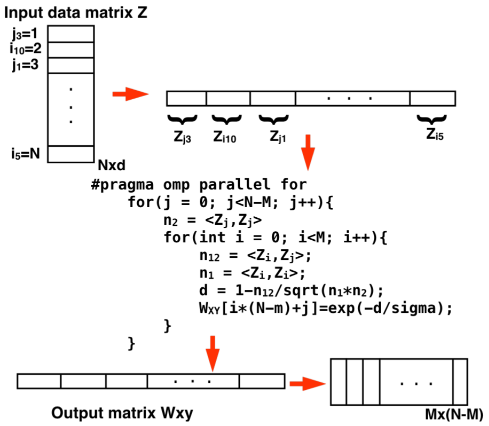
Parallelization of these two classification algorithms involves a parallel for. It is critical to further optimize the OpenMP implementation to get nearly ideal scaling. We detail this process using more complex features of OpenMP such as SIMD and vectorization. Then we use the uniform sampling and chunk of data to get the best performance.

We consider the data set, described in more detail in [13], composed of hyperspectral video sequences recording the release of chemical plumes at the Dugway Proving Ground. We use the 329 frames of the video. Each frame is a hyperspectral image with dimension $128 \times 320 \times 129$, where 129 is the dimension of the channel of each pixel. The total number of pixels is 13,475,840. Since we are dealing with very large data set we choose binary form for smaller storage space and faster I/O. Our test data is 13.91 GB in binary form and the I/O is 36.8× faster than the txt format when testing on Cori Phase I.

We conduct our experiments on single nodes of systems at the National Energy Research Scientific Computing Center (NERSC). Cori Phase I is the newest supercomputer system at NERSC. The system is a Cray XC based on the Intel Haswell multi-core processor. Each node has 128 GB of memory and two 2.3 GHz 16-core Haswell processors. Each core has its own L1 and L2 caches, with 64 KB (32 KB instruction cache, 32 KB data) and 256 KB, respectively; there is also a 40-MB shared L3 cache per socket. Peak performance per node is about 1.2 TFlop/s and peak bandwidth is about 120 GB/s. The resultant machine balance of 10 flops per byte strongly motivates the use of BLAS3 like computations.

4.1 OpenMP Parallelization

Analysis with VTune shows that the most time consuming phase of both two classification algorithms is construction of W_{XY} in the Nyström extension procedure. This phase is a good candidate for OpenMP parallelization because each element of W_{XY} can be computed independently. The procedure of calculating W_{XY} is shown in Fig. 1. We form the data in a N by d matrix Z. Each row of Z corresponds to a data point and it's a vector of dimension d. In computation, we store Z in an array in row major. We randomly select M rows to form the sampled data set $X = \{Z_i\}_{i=1}^{M}$. The other rows form the data set $Y = \{Z_j\}_{j=1}^{N-M}$. Then we use the nested for-loop to calculate the values of W_{XY} by the formula (1). We then put the corresponding value in an array which represent the M by $N - M$ matrix W_{XY}.



```
#pragma omp parallel for
    for(j = 0; j<N-M; j++){
        n_2 = <Z_j,Z_j>
        for(int i = 0; i<M; i++){
            n_12 = <Z_i,Z_j>;
            n_1 = <Z_i,Z_i>;
            d = 1-n_12/sqrt(n_1*n_2);
            W_XY[i*(N-m)+j]=exp(-d/sigma);
        }
    }
```

Fig. 1. The procedure of calculating W_{XY}:

Reordering Loops We have tested re-ordering loops as a means to optimize the algorithm. With analysis, we notice the j-loop is far larger than the i-loop. There are still two ways to do the parallelization. One way is to parallelize the j-loop as inner loop and the other way is to parallelize the j-loop as outer loop. We tried both ways and compared the results.

Step A: Parallelizing the inner j-loop

```
for $i = 0; i < M; i++$
    $n_1 =< Z_i, Z_i >$
    #pragma omp parallel for
    for $j = 1 : N - M$
        $n_{12} =< Z_i, Z_j >$
        $n_2 =< Z_j, Z_j >$
        $d = 1 - n_{12}/\sqrt{n_1 \cdot n_2}$
        $W_{XY}(i, j) = exp(-d/\sigma)$
    end
end
```

Step B: Parallelizing the outer j-loop

```
#pragma omp parallel for
for $j = 1 : N - M$
    $n_2 =< Z_j, Z_j >$
    for $i = 1 : M$
        $n_{12} =< Z_i, Z_j >$
        $n_1 =< Z_i, Z_i >$
        $d = 1 - n_{12}/\sqrt{n_1 \cdot n_2}$
        $W_{XY}(i, j) = exp(-d/\sigma)$
    end
end
```

The results show that parallelizing the outer j-loop is much faster. The run time decreases by a factor of 7. This is because on Cori, each core has its own L1 and L2 cache. When parallelizing the outer j-loop, all the X_is can be read and reside on the L2 of each core and can be used repeatedly. If instead we parallelize the inner j-loop, there are more reads of the X_i and thus the calculation takes more time. Parallelizing the outer j loop also means each thread has more work to do, since the inner i-loop is also part of the j-loop. In this way less overhead and more load balance can be achieved. While if we parallelize the inner j-loop, not only each thread has less work and large load imbalance, but also there are multiple times of thread creation and overhead.

Chunk and Vectorization We further optimize the OpenMP parallelization using vectorization. First, we notice, the norms of Z_is are computed repeatedly in the i-loop. So, we normalize all the Z_is in the previous step, calculating W_{XX}, and store all the normalized Z_is in a new matrix X_{mat}. Then we can calculate the inner product of each Z_j and all the Z_is (X_{mat}) all at once. This make use of BLAS2 instead of the previous BLAS1. Also, we can vectorize the loop when calculating W_{XY}. This optimization reduce the run time of calculating W_{XY} by a factor of 3.

Step C: Calculating W_{XY}, normalize and form all Z_is to X_{mat}

```
#pragma omp for
for $j = 1 : N - M$
    $n_2 =< Z_j, Z_j >$
    $n_{vec} = 1- < X_{mat}, Z_j > /\sqrt{n2}$
    #pragma omp simd aligned
    for $i = 1 : M$
        $W_{XY}(i, j) = exp(-n_{vec}/\sigma)$
    end
end
```

The Nyström extension algorithm is based on a random partition of the whole dataset Z into two disjoint data sets X and Y, where $X = \{Z_i\}_{i=1}^{M}$

and $Y = \{Z_j\}_{j=1}^{N-M}$ and $M << N$. Assuming we can uniformly partition the dataset, so that Z_is are evenly distributed, we can form chunks of Z_js to matrix and further optimize this calculation. The procedure is shown in Fig. 2.

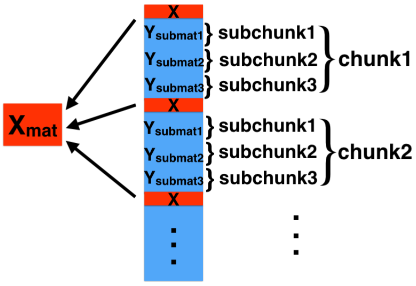

X
Y_{submat1} subchunk1
Y_{submat2} subchunk2
Y_{submat3} subchunk3
chunk1
X
Y_{submat1} subchunk1
Y_{submat2} subchunk2
Y_{submat3} subchunk3
chunk2
X
X_{mat}

Fig. 2. Uniform sampling and dividing Y into chunks and sub-chunks

First, when calculating W_{XX}, we evenly sample Z_is and normalized them. We form the normalized Z_is to a matrix X_{mat}. Then all the data in between two consecutive Z_is are the chunk of Z_js. Since the chunk size is still very large, we further decompose each Y-chunk into sub-chunks. There are several considerations for choosing the sub-chunk size. If it is too small, we waste potential of combining expensive operations. If it is too large, the sub-chunk may run out of lower level cache and needs to be put into the higher cache levels, up to the point where they spill over into DRAM which may cause a substantial performance hit. The optimal value depends on the cache hierarchy, their respective sizes, their latency and so on. For a different architecture, one may consider choosing another value. We pick the the $subchunksize = 64$ when running the codes on Cori Phase I and it can be further optimized.

Then for each sub-chunk, we calculate the Euclidean norm of each row and store them in a vector $n2_{vec}$. This calculation can be vectorized since calculating the norm of each row is independent. We further divide the norms by 1. We then calculate the matrix multiplication $X_{mat} \cdot Y_{submat}$ using BLAS3 function DGEMM. The result is a $m \times subchunksize$ matrix $n12_{mat}$. It is the result of all the inner product of rows in X_{mat} and rows in Y_{submat}. Then we can vectorize the final calculation of values in W_{XY}.

Step D: Calculating W_{XY} using uniform sampling and chunked Y matrices

```
#pragma omp for collapse(2)
for $ychunk = 0; ychunk < m; ychunk++$
    for $j = chunkstart; j < chunkstop; j += subchunksize$
        #pragma omp simd aligned
        for $k = 0; k < subchunksize; k++$
            $n2_{vec}[k] = < Z_{j+k}, Z_{j+k} >$
            $n2_{vec}[k] = 1/\sqrt{n2_{vec}[k]}$
        end
        $n12_{mat} = < X_{mat}, Y_{submat_j} >$
        #pragma omp simd aligned
        for $i = 0; i < m; i++$
            for $k = 0; k < subchunksize; k++$
            $d = 1 - n12_{mat}[i,k] \cdot n2_{vec}[k]$
            $W_{XY}(i, j+k) = exp(-d/\sigma)$
        end
    end
end
```

In this uniform sampling, the chunk size is defined as $chunksize = floor(N/M)$. When M is not divisible by N, the last chunk is larger than the other chunks. Also, *subchunksize* may not be divisible by *chunksize*. So the size of the last subchunk in each chunk needs to be adjusted. The procedure of uniform sampling gives good results as compared to the random sampling and further improves the performance by a factor of 1.7.

We also consider the effect of thread affinity. We choose the thread affinity setting as “scatter”, because it uses one hardware thread per core. While if we use the thread affinity setting to be “compact”, it uses both hardware threads per physical core, leaving one socket idle, which affects scaling performance.

We examined OpenMP thread scaling on a single node of Cori Phase I. The run time decrease and scaling results of different steps of optimizing the OpenMP parallelization are shown in Fig. 3. In Fig. 3 (A), we show the significant speed up of the Nyström loop part. In Step A, in addition to parallelizing the Nyström loop, we also use BLAS3 optimization on the graph MBO algorithm. Since we use BLAS and LAPACK in the serial part of Nyström algorithm and the graph MBO algorithm, their run time also decrease when using multi-cores. In Fig. 3 (B), almost ideal scaling results are achived. Each Cori Phase 1 node has two sockets (NUMA domain) and each socket has 16 cores. Although the absolute performance increases when using more than 16 threads on a single node, NUMA effect is observed that the scaling slows down due to remote memory access to a far NUMA domain.

4.2 Arithmetic Intensity and Roofline Model

Arithmetic intensity is the ratio of floating-point operations (FLOP’s) performed by a given code (or code section) to the amount of data movement (Bytes) that are required to support those operations. Arithmetic intensity in conjunction with the Roofline Model [14] can be used to bound kernel performance and qualify performance in a manner more nuanced than percent-of-peak. Fig. 4 shows the result of using the Roofline Toolkit [15] to characterize the performance of a Cori Phase I node (full 32 cores). The resultant lines (“ceilings”) are bounds on performance. Clearly, in order to attain high performance, one must design algorithms that deliver high arithmetic intensity. In order to characterize the Nyström loop, we used Intel’s Software Development Emulator Toolkit (SDE) to record FLOP’s and Intel’s VTune Amplifier to collect data movement when running on 32 cores of a Cori Phase I node [16, 17]. We can then compare the results to a theoretical estimate based on the inherent requisite computation and data movement.

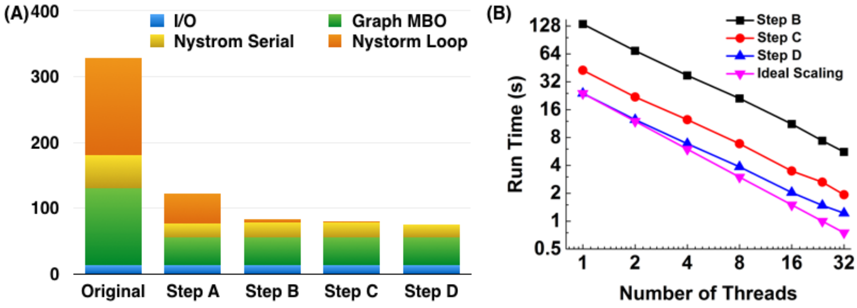


Fig. 3. (A): The run time of different optimization steps. Step A: parallelizing the inner j-loop and BLAS3 optimization on Graph MBO. Step B: parallelizing the outer j-loop. Step C: normalizing and forming all Z_is to X_{mat}. Step D: using uniform sampling and chunked Y matrices. (B): The scaling results of the OpenMP parallelization of the Nyström loop. The black line with squares, the red line with circles and the blue line with triangles show the scaling results of step B, C and D respectively. The pink line with upside down triangles shows the ideal scaling.

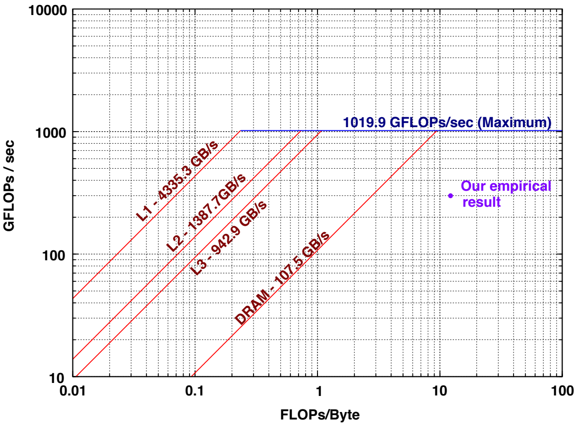


Fig. 4. Empirical Roofline Toolkit results for a Cori Phase I node. Observe, DRAM bandwidth constrains performance for a wide range of arithmetic intensities.

As shown in Fig. 1, the memory access has two major components — one must read data from the matrix Z from DRAM and then write the results in to a matrix W_{XY}. The size of data matrix Z is $N \times d$, where $N = 13,475,840$ and $d = 129$ for our test data. As we store the data in double precision, the total size of the matrix (and hence volume of data read) is 13.907×10^9 bytes. In the inner loop, the processor must continually access M rows of the matrix

Z. As the resultant volume of data (103,200 bytes) easily fits in cache, we need only real each Z_i once (data movement is well proxies by compulsory cache misses). The size of the matrix W_{XY} is $(N - M) \times M$, where $M = 100$. As each double-precision element is written once, we can bound write data movement as $(N - M) \times M \times 8 = 10.78 \times 10^9$ bytes. A similar calculation can be performed to calculate the requisite number of floating-point operations. In the optimized code, although there are dot products for $< Z_j, Z_j >$ coupled with a reciprocal square root and one exponential per element of W_{XY}, the DGEMM used for calculating $X_{mat} \times Y_{submat}$ should dominate the flop count. The matrix X_{mat} is 100×129, the matrix Y_{submat} is on average 64×129, and there are roughly $13,475,840/64 = 210560$ Y_{submat} matrices. Thus, the number of floating-point operations in the loop is about $210560 \times 2 \times 64 \times 129 \times 100 = 347.68 \times 10^9$ (ignoring any BLAS2 operations in DGEMM, the dot products, and exponential).

	Theoretical	Empirical
Bytes Read	13.907×10^9	17.123×10^9
Bytes Written	10.781×10^9	12.256×10^9
FP operations	$>347.68 \times 10^9$	385.59×10^9
Arithmetic Intensity (flop:byte)	>14.1	13.12

Table 1. Theoretical estimates and Empirical measurements (using VTune and SDE) of data memory and floating-point operations for the Nyström loop.

Table 1 presents our theoretical estimates and empirical measurements (using VTune and SDE) of data memory and floating-point operations for the Nyström loop. Generally speaking our rough theoretical model slightly underestimated each quantity. Multiple sockets (each with their own caches) may be required to read unique bytes, but in reality will access overlapping data due to the realities of large cache lines and hardware stream prefetchers. In terms of floating-point operations it is clear DGEMM (the basis for our theoretical model) constitutes over 90% of the total flop count. Overall, with a run time of about 1.28 seconds, the optimized code attains about 300GFlop/s of performance and 22GB/s of DRAM bandwidth at an arithmetic intensity of just over 13 flops per byte. At such a high arithmetic intensity, Figure 4 suggests DRAM bandwidth will not be the ultimate limiting factor, but further optimizations for the cache hierarchy (coupled with NUMA optimizations) may be in order.

5 Conclusion and Future Work

In this paper, we present a parallel implementation of two novel classification algorithms using OpenMP. We show OpenMP parallel and simd regions in combination with optimized library routines achieving almost ideal scaling and manyfold speedup over serial implementations. We are currently working on KNL "white boxes" (pre-release hardware) and we will include those results and necessary OpenMP optimizations in the final paper as allowed under our NDA with Intel. (Expected release is before the final paper, if accepted, is due.)

6 Acknowledgments

This work was supported by NSF grants DMS-1417674 and DMS-1045536 and AFOSR MURI grant FA9550-10-1-0569. We would like to thank Dr. Da Kuang for his suggestions on optimizing the serial codes. This work was also supported by U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

1. Meng, Z., Merkurjev, E., Koniges, A., Bertozzi, A.L.: Hyperspectral Video Analysis Using Graph Clustering Methods. Image Processing On Line, submitted
2. Stoer, M., Wagner, F.: A simple min-cut algorithm. Journal of the ACM (JACM) 44.4 : 585-591 (1997)
3. Szlam, A., Bresson, X.: A total variation-based graph clustering algorithm for cheeger ratio cuts. UCLA CAM Report : 09-68 (2009)
4. Bertozzi, A.L., Flenner A.:Diffuse Interface Models on Graphs for Classification of High Dimensional Data. SIAM Review, 58(2), pp. 293-328 (2016)
5. Chung, F.: Spectral graph theory. Vol. 92. American Mathematical Soc., (1997)
6. Von Luxburg, U.: A tutorial on spectral clustering. Statistics and computing 17.4: 395-416 (2007)
7. Van Gennip, Y., Bertozzi, A. L. *Gamma*-convergence of graph Ginzburg-Landau functionals. Advances in Differential Equations 17.11/12: 1115-1180 (2012)
8. Bertozzi, A.L., Flenner, A.: Diffuse interface models on graphs for classification of high dimensional data. Multiscale Modeling & Simulation 10.3: 1090-1118 (2012)
9. Fowlkes, C., Belongie, S., Chung, F., Malik, J.: Spectral grouping using the Nyström method. Pattern Analysis and Machine Intelligence, IEEE Transactions on 26.2: 214-225 (2004)
10. Merkurjev, E. and Kostic, T., Bertozzi, A.L.: An MBO scheme on graphs for classification and image processing. SIAM Journal on Imaging Sciences 6.4: 1903-1930 (2013)
11. Hu, H., Sunu, J., Bertozzi, A.L.: Multi-class graph Mumford-Shah model for plume detection using the MBO scheme. Energy Minimization Methods in Computer Vision and Pattern Recognition. Springer International Publishing (2015)
12. Demmel, J.W.: Applied numerical linear algebra. Siam (1997)
13. Broadwater, J. B., Limsui, D., Carr, A. K.: A primer for chemical plume detection using LWIR sensors. Technical Paper, National Security Technology Department, Las Vegas, NV (2011)
14. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Communications of the ACM 52.4: 65-76 (2009)
15. Roofline Toolkit: https://bitbucket.org/berkeleylab/cs-roofline-toolkit
16. Intel Software Development Emulator: https://software.intel.com/en-us/articles/intel-software-development-emulator
17. Doug Doerfler, Understanding Application Data Movement Characteristics using Intel VTune Amplifier and Software Development Emulator tools, Intel Xeon Phi User Group (IXPUG) (2015)
18. Intel VTune Official Website: https://software.intel.com/en-us/intel-vtune-amplifier-xe