Optimization and parallelization of a time series classification algorithm

Robert S. Sinkovits

San Diego Supercomputer Center, email: sinkovit@sdsc.edu

***Abstract* – This technical report describes the steps taken to optimize and parallelize a time series classification algorithm as part of an Extended Collaborative Support Services (ECSS) project with XSEDE researcher Ramon Huerta at the University of California, San Diego. Switching from the GNU compiler to the Intel compiler and enabling Advanced Vector Extensions (AVX) resulted in a 2x speedup, while linking to the Intel Math Kernel Library (MKL) instead of the default LAPACK library further improved performance and provided an easy path to thread-level parallel execution. These changes resulted in a combined 46x speedup relative to a single core when running on all 16 cores of a dual-socket Intel Sandy Bridge node. Parallelization of several loops using OpenMP directives and the removal of an unnecessary duplicate call to a computationally demanding routine brought the total speedup to 86x. Optimization of linear algebra operations using time-space tradeoffs ultimately resulted in a total speedup of 168x relative to the original version and build of the code.**

***Keywords* – optimization, parallelization, MKL, time-space tradeoff, time series algorithm.**

1 INTRODUCTION

The XSEDE [1] Extended Collaborative Support Services (ECSS) program pairs PIs (i.e. recipients of awards on compute and storage resources allocated through XSEDE) with XSEDE computational science and technology experts for collaborations lasting up to one year. The goal of ECSS is to improve the productivity of the PIs through a range of activities including software optimization and parallelization, advanced visualization, workflow development, deployment of science gateways, and porting software to new architectures.

We describe here the results of a collaboration spanning 2012-2013 that addressed improving the performance of a time series classification algorithm. This work was carried out under the Extended Support for Research Teams (ESRT) area of ECSS.

At the time of this collaboration the project PI, Dr. Ramon Huerta was a Research Scientist with the BioCircuits Institute at the University of California, San Diego. He is currently a Principal Machine Learning Scientist at Amazon. He holds degrees in physics and engineering and has worked primarily in computational neuroscience and non-linear dynamics.

For background and completeness, the PI statement for the ECSS project is reproduced below. This has been very lightly edited for clarity and can be skipped for readers who are only interested in the technical details of optimization and parallelization.

PI statement: We have developed classification calibrated algorithms that are faster to train than regular support vector machines. The trick that we discovered by exploring a continuous set of loss functions is that we can modify the symmetric structure of the loss function for the negative and positive examples, avoiding the use of a regularization term during training. This loss function is classification calibrated for an infinite margin for any number of classes available in the training dataset. Moreover, the classification function approaches de Bayes decision boundary, which is an ideal property to fulfill for any classifier. For the implementation of the algorithm we have used the guidance and code optimization provided by Robert Sinkovits at the San Diego Supercomputer Center (SDSC) to take advantage of the areas of the code where parallelization was possible using OpenMP and the Intel Math Kernel Libraries (MKL) [2] on *Gordon* [3]. We tested the code using the SDSC resources on a set of 14 public databases, and we have found significant advantages in training the models. The average speed gained by the infinite margin method was about 4-fold. Every second of computation of the comparative methods only took half a second in the rest of the methods. Finally, the support by XSEDE allocation ASC120025 has provided the means to fully complete the experimental section of the paper entitled: "On the practical implications of a continuous family of calibrated and uncalibrated multiclass loss functions" that will be submitted for consideration by the *Journal of Machine Learning*. The experimental section was carried out on 14 datasets of different levels of complexity. In order to measure the classification accuracy, we generate 100 different partitions of each dataset except for four of the simplest datasets, in which we create 1000 partitions to obtain better average estimations. Specifically, given a data set of size N, N/2 samples are selected as the training subset and the remaining N/2 examples constitute the test set. The training samples are used to explore a 10-fold cross-validation grid of 10x5 meta-parameter values while the test set is used to report a reliable estimation of the performance of the model. These fast training algorithms will be critical when the time series of individual data of the elderly can be collected.

Note on terminology: This technical report was adapted from a final report delivered to the ECSS PI. The function

and variable names used throughout match those used in the original version of the software.

2 SOFTWARE OPTIMIZATION AND PARALLELIZATION

Our focus was on optimizing and parallelizing Ramon Huerta's autoregressive kernel application to reduce time to solution. As originally written, this was a serial application. The executable was built using the GNU C++ compiler (g++) and linked the default version of the LAPACK library. Through a combination of compiler, compiler option and library changes together with minor modifications to accommodate more recent LAPACK calling conventions, we were able to achieve a speedup of 46x relative to the original code on a single *Gordon* compute core. A further 3.6x compounded speedup was realized through substantive source code changes, resulting in a total speedup of 168x. We had originally anticipated developing a distributed memory version of the code using parallel linear algebra libraries, but this was not deemed to be necessary given the dramatic reductions in run times that we obtained using a shared memory approach.

The next five subsections describe the compiler, library and code changes in greater detail. Note that compiler and library versions were not recorded but based on date of work were likely Intel C++ 13.0 or 14.0, GCC 4.6 or 4.7 and MKL 10.0 or 11.0. Run times and relative performance gains at each step are summarized below.

Table 1. Summary of performance improvements for ARsvm code. All timings obtained on *Gordon* and speedups are relative to original serial code run on a single *Gordon* core. Benchmarks used parameters and input files provided by Huerta lab: -C 1.0 -T 0.01 -p 100 -i filter_train.0.dat -I filter_test.0.dat. Run times for other conditions (-C 100.0) are approximately 75% lower.

Notes	Cores	t (h:m:s)	Speedup
Original code, GNU compiler	1	11:22:00	-
Intel compiler and enable AVX	1	05:41:49	2.0
Link threaded MKL library	16	00:14:46	46.2
OpenMP directives in key loops	16	00:13:10	52.5
Remove duplicate function call	16	00:07:58	85.6
Optimization of linear algebra	16	00:04:04	167.7

2.1. Use Intel compiler and enable AVX

We encountered no difficulties in porting the software from Huerta's local compute resources to SDSC's *Gordon* supercomputer. The relevant setting in the original and modified Makefile are shown below

```
# Original
CXX = g++
CXXFLAGS= -O3  -Wall

# Modified
CXX = icpc
CXXFLAGS= -O3 -xHOST -no-prec-div -Wall
```

The -xHOST flag instructs the compiler to use the highest-level instruction set that is available, which in the case of *Gordon* is AVX. Since we were still using the default LAPACK libraries at this point and the code spends most of its time in linear algebra routines, the speedup of 2x obtained by enabling AVX is exactly what would be expected since these routines tend to obtain a large fraction of peak performance on both AVX (eight flops/cycle/core) and non-AVX (four flops/cycle/core) processors.

2.2. Link MKL and use thread-level parallelism

The Intel MKL routines have been specifically tuned for optimal performance on Intel hardware. The MKL routines are also threaded, so going from serial execution to shared-memory parallelism is trivial. Minor changes to the source code were needed, though, in order to use the MKL.

(1) Long integer types must be changed to regular integers when calling MKL. These changes must also be propagated through the user functions that contain MKL calls.

```
// Original
void multXI(…, long NN, long l, double *xy, …

// Modified
void multXI(…, int NN,  int l, double *xy, …
```

(2) Calls to MKL from C do not require name mangling (i.e. adding of underscores or other decorators to function names).

```
// Original
dgemm_(…);

// Modified
dgemm(…);
```

(3) Some calls had to be modified to use the standard argument list. This may involve passing pointers instead of values and moving return values inside the argument list

```
// Original
info=dgetri(len, C, len, ipiv, work, lwork);

// Modified
dgetri(&len, C, &len, ipiv, work, &lwork, &info);
```

(4) Headers were added for MKL and OpenMP support

```
#include <omp.h>
#include <mkl.h>
#include <mkl_cblas.h>
```

(5) OpenMP compiler flag and links to MKL added to Makefile

```
# Original
CXX = g++
CXXFLAGS= -O3  -Wall
LDFLAGS=-lm -llapack -lblas

# Modified
CXX = icpc
CXXFLAGS= -O3 -xHOST -no-prec-div -fopenmp –Wall
LDFLAGS = -lm -mkl
```

(6) Set number of cores and threads in job submission file. Example below is taken from *Gordon* batch script using the TORQUE/PBS resource manager.

```
#PBS -l nodes=1:ppn=16:native
export OMP_NUM_THREADS=16
```

2.3. Apply OpenMP directives to key loops

The original version of the code spent most of its time in the linear algebra routines DGEMM and DSYRK. After switching to MKL and running in parallel, the relative amount of exclusive time spent in the two user routines kAR and kARtest increases. Adding OpenMP directives resulted in modest (~7%) reductions in run time. The examples below are from kAR. Note that the loop indices are declared as private, but this is done by default according to the OpenMP standard.

```
#pragma omp parallel for private(k)
  for(k=0;k<dimx;++k)
    x[i][dimx+k]=x[j][k];

#pragma omp parallel for private(k,k2)
  for(k=0;k<Mp;++k) {
    for(k2=0;k2<k;++k2)
      Inv[k+k2*M*p]=Inv[k2+k*M*p];
  }
```

2.4. Remove duplicate function call

The original version of the code contained an unnecessary duplicate call to kARtest in function simplerrorrate. Making the call just once and reusing the result reduces the run time from roughly 13 minutes to eight minutes.

```
// Original
printf("K=%lf\n",kARtest(i,j));fflush(stdout);
Ei+=ASV[j]*kARtest(i,j);

// Modified
double kARtest_result = kARtest(i,j);
printf("K=%lf\n",kARtest_result);fflush(stdout);
Ei+=ASV[j]*kARtest_result;
```

In this example, one might assume that the compiler could recognize this optimization opportunity, but this cannot be done automatically since it's not known at compile time if the function has side effects. Using our knowledge of the code, we are able to safely implement this change.

2.5. Optimize linear algebra

After implementing the changes described above, the code spends nearly all of its time in calls to DSYRK (double precision rank k symmetric update)

$$(1)\quad C \coloneqq \alpha AA^T + \beta C$$

where α and β are constants, A and C are arrays, and A^T is the transpose of A. The following call trees show how DSYRK is accessed from the user-defined functions method2 and simplerrorrate:

```
method2 → kAR → multpdblas → DSYRK
simplerrorrate → kARtest → multpdblas → DSYRK
```

By implementing a time-space tradeoff and making changes that cascade through the call tree, the time spent in DSYRK can be reduced by nearly half.

We start by expressing the N x M dimensional matrix A as the concatenation of two N x M/2 arrays as illustrated below using a 4 x 4 matrix as a concrete example.

$$(2)\quad \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} & c_{13} & c_{14} \\ b_{21} & b_{22} & c_{23} & c_{24} \\ b_{31} & b_{32} & c_{33} & c_{34} \\ b_{41} & b_{42} & c_{43} & c_{44} \end{bmatrix} = [B \quad C]$$

The operation AA^T can then be written as the sum of two matrix multiplications involving B and C, the two halves of the original array.

$$(3)\quad AA^T = [B \quad C]\,[B \quad C]^T = [B \quad C] \begin{bmatrix} B^T \\ C^T \end{bmatrix} = BB^T + CC^T$$

Next, we note that the arrays that are operated on by DSYRK have the form just described. For example, in method2 the indices i and j are passed to kAR where the elements offset dimx positions from the address x[i][0] are updated using the data starting at x[j][0].

```
for(k=0;k<dimx;++k)
  x[i][dimx+k]=x[j][k];
```

The DSYRK operation is then carried out using the 2*dimx elements starting at x[i][0]. The first and second sets of dimx elements starting at x[i][0] correspond to the arrays B and C in Eqns. 2 and 3.

Finally, we recognize that the we can simply pre-calculate the DSYRK results for the sets of dimx elements starting at x[0][0], x[1][0], … x[N-1][0] and replace the DSYRK operations on arrays of size N x M with a matrix addition on arrays of size N x M/2.

In the original implementation of method2 a total of N(N-1)/2 calls to DSYRK are made indirectly through kAR and multpdblas. In the modified version of the code an array of N pre-calculated DSYRK results on the half-sized arrays are instead passed to kAR where the matrix additions are performed

```
// Original (simplified code from method2)
for(int i=0;i<N;++i) {
  for(j=i;j<N;++j) {
    K[j][i]=K[i][j]=kAR(i,j);

// Modified (simplified code from method2)
for(int i=0;i<N;++i) {
  for(j=i;j<N;++j) {
    K[j][i]=K[i][j]=kAR(i,j,precalc);
```

Similar changes are made to simplerrorrate, except in this case two sets of pre-calculated results need to be passed to kARtest since the array operations involve the arrays x and SVALLX.

```
// Modified (simplified codefrom simplerrorrate)
for(int i=0;i<N;++i) {
  for(unsigned int j=0;j<ASV.size();++j) {
```

```
double kARtest_result =
    kARtest(i,j,precalc,precalc2);
```

3 DISCUSSION

Through a combination of compiler, compiler flag, and library changes together with a small set of code modifications, we were able to reduce the run time for a typical computationally intensive time series classification problem from nearly 11 and 1/2 hours to just over four minutes. Although the last change (see section 2.5) involves a time-space tradeoff and increases the memory footprint of the application, it still requires only a few GB of memory for a typical problem.

In addition to delivering improved software and knowledge transfer to Ramon Huerta and his lab, the ECSS project also reinforced a few of the common lessons of code optimization:

1. Focus on the low-hanging fruit first, such as using the best compiler (Intel or PGI rather than GNU), compiler flags, and vendor-tuned libraries (e.g. Intel's MKL).
2. Re-profile code after optimizing since new hot spots may have emerged.
3. Go beyond profiling and look at what the code is doing. This may identify opportunities to eliminate unnecessary work or reformulate the algorithms in a more efficient manner

4 ACKNOWLEDGMENTS

This work was supported by NSF awards OAC-1053575 (XSEDE: eXtreme Science and Engineering Discovery Environment) and ACI-1548562 (XSEDE 2.0: Integrating, Enabling and Enhancing National Cyberinfrastructure with Expanding Community Involvement). The deployment and operation of the *Gordon* supercomputer was supported by NSF grant OAC-0910847. The author thanks Ramon Huerta for many useful discussions and his participation in the ECSS program.

5 REFERENCE

[1] Towns, J., T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G.D. Peterson, R. Roskies, J.R. Scott, and N. Wilkins-Diehr, *XSEDE: accelerating scientific discovery*. Computing in Science & Engineering, 2014. **16**(5): p. 62-74.

[2] Wang, E., Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, *Intel math kernel library*, in *High-Performance Computing on the Intel® Xeon Phi™*. 2014, Springer. p. 167-188.

[3] Strande, S.M., P. Cicotti, R.S. Sinkovits, W.S. Young, R. Wagner, M. Tatineni, E. Hocks, A. Snavely, and M. Norman. *Gordon: design, performance, and experiences deploying and supporting a data intensive supercomputer*. in *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the campus and beyond*. 2012. ACM.