Parallel Distributed Kernel Estimation

Yi-Chen Zhang

Department of Statistics and Probability, Michigan State University

April 4, 2017

Abstract

Nonparametric kernel methods are becoming more commonplace for data analysis, modeling, and inference. Unfortunately, these methods are known to be computationally burdensom. The burden increases as the amount of available data rises and can quickly overwhelm the computational resources present in modern desktop workstations. In this paper, we consider a parallel implementations of a number of popular kernel methods based on the Open Multi-Processing (OpenMP) standard. The benchmark of serial and parallel results will be compared. A simple demonstration indicates how they can dramatically reduce the computational burden often associated with kernel methods thereby achieving an almost ideal parallel speed-up.

Keywords: Nonparametric kernel; OpenMP.

1. Introduction

Kernel methods are widely acknowledged to be a powerful tool for applied data analysis. They have been successfully applied to a large number of problem domains spanning a range of fields including economics, physics, and statistics to name but a few. Unlike many flexible modeling tools such as spline smoothers and neural networks which are typically of computational order $O(nk)$ where n is the number of observations and k is the number of variables, kernel methods are of order $O(n^2k)$. Data-driven methods of bandwidth selection such as cross-validation (Stone, 1974), while widely acknowledged to be an indispensable component of applied data analysis, can add an additional order of computational magnitude to kernel methods. It is not uncommon to encounter researchers for whom the computational burden of applying kernel methods even to moderately sized data sets is precived to be too costly to justify their use.

Fortunately, recent developments in parallel computing are not only capable of dramatically reducing the computational burden associated with kernel methods, but are ideally suited to handling high-dimensional data sets and are surprisingly easy to implement from a programming standpoint. In this paper, we outline one such implementation based on the OpenMP standard, an internationally recognized and freely available parallel computing paradigm that may be used to explicitly direct multi-threaded, shared memory parallelism.

This paper is organized as follows: In Section 2 we give a briefly review of the OpenMP standard, then we proceed to the discussion of the kernel method implemented with OpenMP and provid the pseudo-code in Section 3. In section 4, we examine the performance benchmarks of serial and parallel run of kernel method. Finally, we discuss our results and conclude in Section 5.

2. Open Multi-Processing

We quote from wikipedia for a short introduction. OpenMP is an implementation of multithreading, a method of parallelizing whereby a master thread forks a specified number of slave threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

The section of code that is meant to run in parallel is marked accordingly, with a compiler directive that will cause the threads to form before the section is executed (see Hager and Wellein (2011)). Each thread has an integer id attached to it, and the master thread has an id of 0. After the execution of the parallelized code, the threads join back into the master thread, which continues onward to the end of the program.

By default, each thread executes the parallelized section of code independently. Work-sharing constructs can be used to divide a task among the threads so that each thread executes its allocated part of the code, such features are extremely valuable for the kernel density estimation, which computational costs are considerably high. For more details regarding the OpenMP standard and library, refer to Lawrence Livermore National Laboratory website.

3. Kernel density estimatoin

For what follows, we will restrict attention to an implementatoin written in C/C++ on the kernel density estimator. Consider the kernel density estimator originally proposed by (Rosenblatt, 1956) which is defined as

$$\hat{f}_n(x) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{h_n}K\left(\frac{x - X_i}{h_n}\right) \tag{1}$$

where X_i's, $i = 1, \ldots, n$, are the observation data and x is the evaluation data (typically they are the same), the kernel function $K(z)$ satisfies $\int K(z)dz = 1$ and some other regularity conditions depending on its order, and h_n is a bandwidth satisfying $h_n \to 0$ as $n \to \infty$. Loftsgaarden and Quesenberry (1965) considered replacing the 'fixed bandwidth' found in Rosenblatt's (1956) estimator with nearest neighbor weights around the point x ('variable bandwidth'), while the adaptive kernel estimator with weights depending on X_i rather than x was considered by Breiman et al. (1977) and Abramson (1984) ('adaptive bandwidth'). This latter estimator is obtained by replacing h_n with $h_{n,i}$ in (1).

These three density estimators (fixed, variable, and adaptive bandwidth) are perhaps the most popular of the kernel density estimation methods. Each of these estimators share a common feature—they involve $O(n^2k)$ computations in contrast to, say, parametric density functions that involve only $O(nk)$ computations (in this instance, $k = 1$, however, we shall relax this in a moment).

Consider the following example of C/C++ pseudo-code implementation for an adaptive bandwidth density estimator (Abramson, 1984).

```
/* Compute a kernel density estimate */
for ( i = 0; i < num_of_x; ++i ){
  sum_ker = 0.0;
  for ( j = 0; j < num_of_X; ++j ){
    sum_ker = sum_ker + kernel((x[i]-X[j])/h[i])/h[i];
  }
  pdf[i] = sum_ker/n;
}
```

Without loss of generality, we shall assume for illustrative purposes that $i = 1, \ldots, n$ and $j = 1, \ldots, n$. It is evident from this simple example that, without resorting to approximations, there is no way to avoid the $O(n^2k)$ computations if one wishes to compute a kernel density estimate at each of the sample realizations as is often desired.

Now consider the following OpenMP-enabed pseudo-code which demonstrates how the OpenMP libraries can be disarmingly simple to implement—first one needs to include the header file labeled "omp.h", next one defines compiler directives and then one makes use of a few of the data scope attribute clauses thereby converting the serial code into parallel code.

```
#include <omp.h>
...
#pragma omp parallel num_threads(m) default(none)
                     shared(x,X,i,pdf,h) private(sum_ker,j)
  {
#pragma omp for
  for ( i = 0; i < num_of_x; ++i ){
    sum_ker = 0.0;
    for ( j = 0; j < num_of_X; ++j ){
      sum_ker = sum_ker + kernel((x[i]-X[j])/h[i])/h[i];
    }
    pdf[i] = sum_ker/n;
  }
```

First, note the simplicity of the implementation—two lines of strightfrward OpenMP derivatives and clauses call, one line to define the number of threads and the default scope for all variables in the parallel region. The clause "shared" specifies variables in its list to be shared among all threads and the clause "privated" specifies variables in its list to be private to each thread. The "for" directive specifies that the iterations of the loop immediately following it must be executed in parallel.

Next, note that if the number of threads equals the number of evaluation observations then we have reduced this computation from an $O(n^2k)$ computation to an $O(nk)$ one. In general this will not be the case. Regardless, if the overhead involved with the interface is negligible, we achieve a linear speed-up which can have rather dramatic real-time implications. That is, we have converted the burden of $O(n^2k)$ computations computed on a single processor into one of $O(n^2k/p)$ being borne simultaneously by each of p processors where p is under our control.

4. Performance benchmarks

In this performance benchmarks study, we first check the correctness of the program and then investigate the serial performance of the kernel density estimation. Later we will compare it with the parallel performance.

The data were generated from Normal distribution with mean $\mu = 2$ and standard deviation $\sigma^2 = 0.5$. The white noise with variance 0.1 were added into the data for practical purpose. The density function is estimated by equation (1) based on four different type of kernel functions that are commonly used: Uniform, Epanechnikov, Gaussian, and Quartic kernel. We examine the kernel density estimation method in more detail by overlaying the histogram and the density plots. The results are shown in Figure 1.

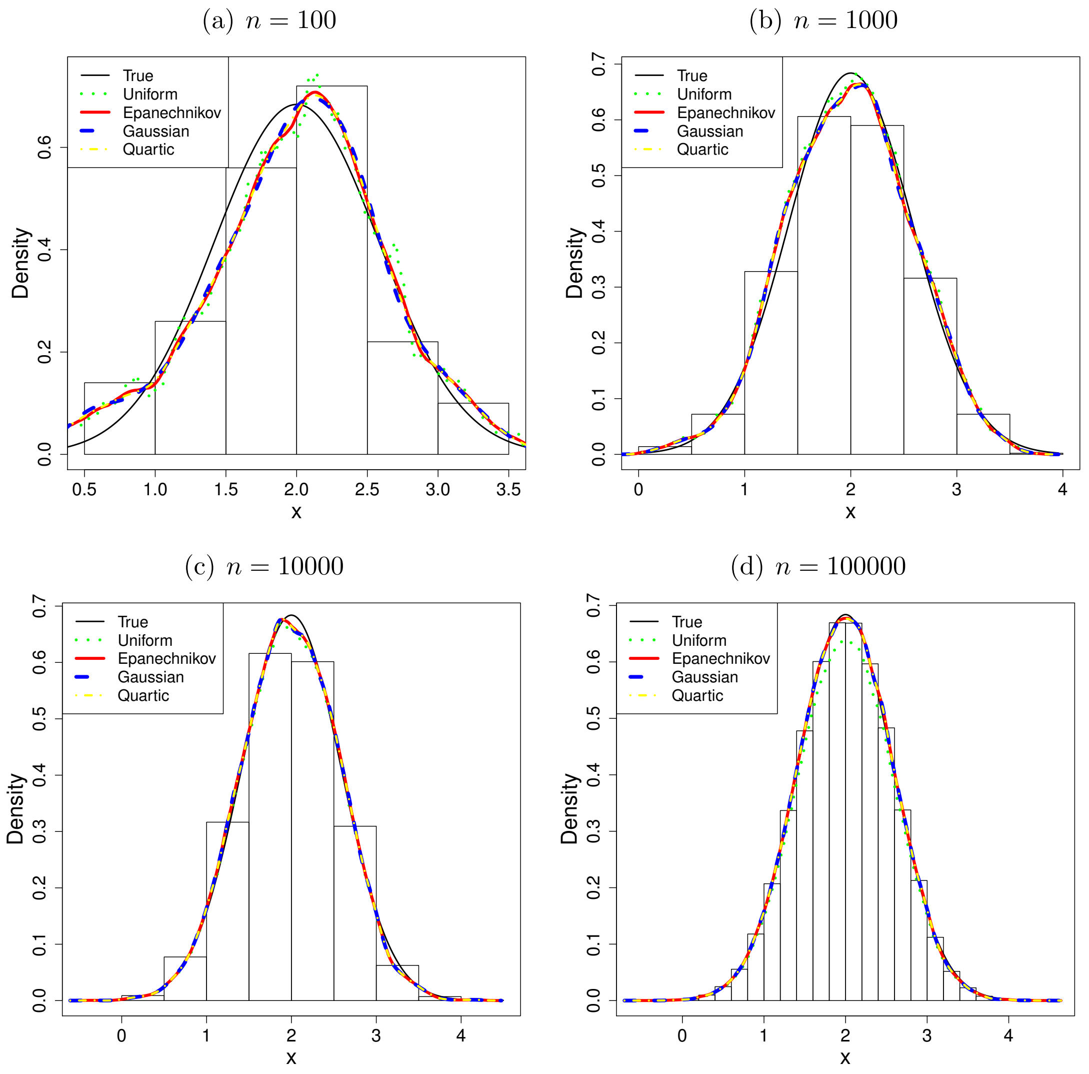


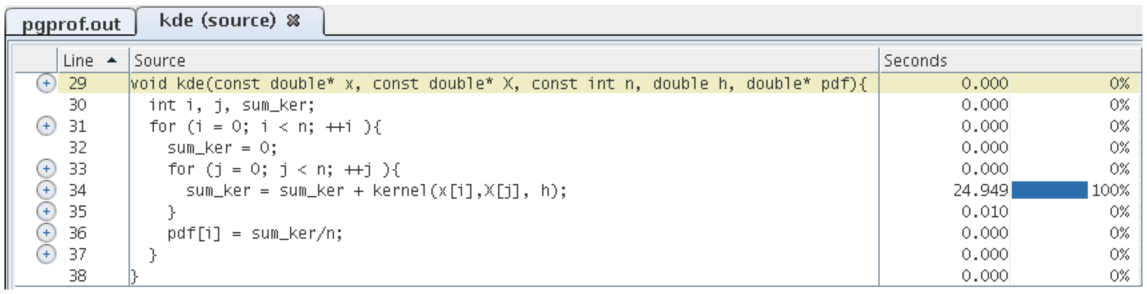
Figure 1: Kernel density estimation with four different types of kernel functions for sample size n from 100 to 100000.

From Figure 1(a) to Figure 1(d), it is clear that the estimated density functions converge to the true density function as sample size n increases. This correctness check leads us to the further investigation on our program performance.

The GNU gprof was used on one serial run of our program and the flat profile results are listed in Table 1. The kernel density estimation function 'kde' takes almost 100% of total execution time. We then ran PGI prof for more detail examination of the 'kde' function. Figure 2 indicates that the 'kde' function spends most of time to calculate kernel values. As we discussed in previous section, we shall consider to parallel our program to save computation time.

Table 1: GNU prof result: each sample counts as 0.01 seconds.

%	cumulative	self		self	total	
time	seconds	seconds	calls	Ts/call	Ts/call	name
99.97	25.37	25.91	1	25.09	25.09	kde
0.02	25.37	0.00	3	0.00	0.00	vector
0.00	25.37	0.00	1	0.00	0.00	rnorm
0.00	25.37	0.00	1	0.00	0.00	runif
0.00	25.37	0.00	1	0.00	0.00	linspace
0.00	25.37	0.00	1	0.00	0.00	operator+



```
Line  Source                                                                   Seconds
29    void kde(const double* x, const double* X, const int n, double h, double* pdf){    0.000      0%
30      int i, j, sum_ker;                                                       0.000      0%
31      for (i = 0; i < n; ++i ){                                                0.000      0%
32        sum_ker = 0;                                                           0.000      0%
33        for (j = 0; j < n; ++j ){                                              0.000      0%
34          sum_ker = sum_ker + kernel(x[i],X[j], h);                           24.949    100%
35        }                                                                      0.010      0%
36        pdf[i] = sum_ker/n;                                                    0.000      0%
37      }                                                                        0.000      0%
38    }                                                                          0.000      0%
```

Figure 2: PGI prof result.

We perform serial and parallel benchmarks on a 1-node with 28-processors Laconia (dev-intel16) cluster running CentOS 6.6 using the OpenMP implementation. Code was compiled with g++ version 6.2.0, and four different kernel functions were used. We report the execution time for an x-processors ($x = 1, 2, 4, 8, 10, 16, 20, 24$) using sample sizes of $n = 10^2$ through $n = 10^5$. The execution time was averaged over 100 program replications. The timing of serial and parallel benchmarks are reported in Table 2.

Table 2: Execution time (in seconds) for kernel density estimation on a x-processors

Kernel	n	Number of processors								
		1	2	4	8	10	16	20	24	
Uniform	10^2	1.378	0.759	0.420	0.271	0.545	0.514	0.514	0.503	$(\times 10^{-7})$
	10^3	1.443	0.729	0.397	0.202	0.159	0.100	0.080	0.067	$(\times 10^{-4})$
	10^4	1.382	0.713	0.375	0.198	0.157	0.097	0.078	0.065	$(\times 10^{-1})$
	10^5	1.373	0.715	0.373	0.198	0.156	0.097	0.077	0.064	$(\times 10^{2})$
Epanechnikov	10^2	2.380	1.212	0.720	0.425	0.583	0.298	0.269	0.611	$(\times 10^{-7})$
	10^3	2.424	1.213	0.689	0.339	0.270	0.170	0.135	0.115	$(\times 10^{-4})$
	10^4	2.426	1.216	0.640	0.340	0.269	0.168	0.134	0.112	$(\times 10^{-1})$
	10^5	2.397	1.231	0.641	0.340	0.270	0.169	0.135	0.112	$(\times 10^{2})$
Gaussian	10^2	4.596	2.612	1.211	0.713	0.992	0.893	0.635	0.581	$(\times 10^{-7})$
	10^3	4.799	2.363	1.236	0.655	0.528	0.329	0.265	0.220	$(\times 10^{-4})$
	10^4	4.591	2.332	1.240	0.660	0.522	0.371	0.263	0.218	$(\times 10^{-1})$
	10^5	4.589	2.344	1.221	0.655	0.522	0.352	0.262	0.234	$(\times 10^{2})$
Quartic	10^2	4.616	2.073	1.159	0.668	0.541	0.629	0.707	0.626	$(\times 10^{-7})$
	10^3	4.326	2.130	1.069	0.585	0.465	0.297	0.233	0.196	$(\times 10^{-4})$
	10^4	4.079	2.085	1.076	0.583	0.465	0.290	0.233	0.194	$(\times 10^{-1})$
	10^5	4.148	2.099	1.117	0.587	0.465	0.291	0.232	0.193	$(\times 10^{2})$

We can see from Table 2 that, for the samller sample size of $n = 100$, there are some slight overheads which become more marked as the number of processors increase. Figure 3(a) clearly indicates that the overheads happend for more number of processors (10, 16, 20, 24) is used. However, as the sample size increases quickly disappear (see Figure 3(b) to 3(d)). Moreover, we observe that we indeed obtain a speed-up that is linear in the number of available processors—a doubling of the number of processors leads to a halving of the execution time. For example, a perfect speed-up would result in a $\frac{1}{24}$ or 0.042 ratio of execution time for 24-processors realtive to 1-processor, while it is clear that we quickly approach this ratio as n increases.

A few words on diminishing returns to adding more processors are in order. A careful examination of Table 2 reveals that as additional processors are added, the marginal reduction in computing time dissipates (i.e. one obtains a larger reduction when going from two to four processors than when going from 20-24 processors). In fact, it is possible for run-time to actually deteriorate (increase) when the additional

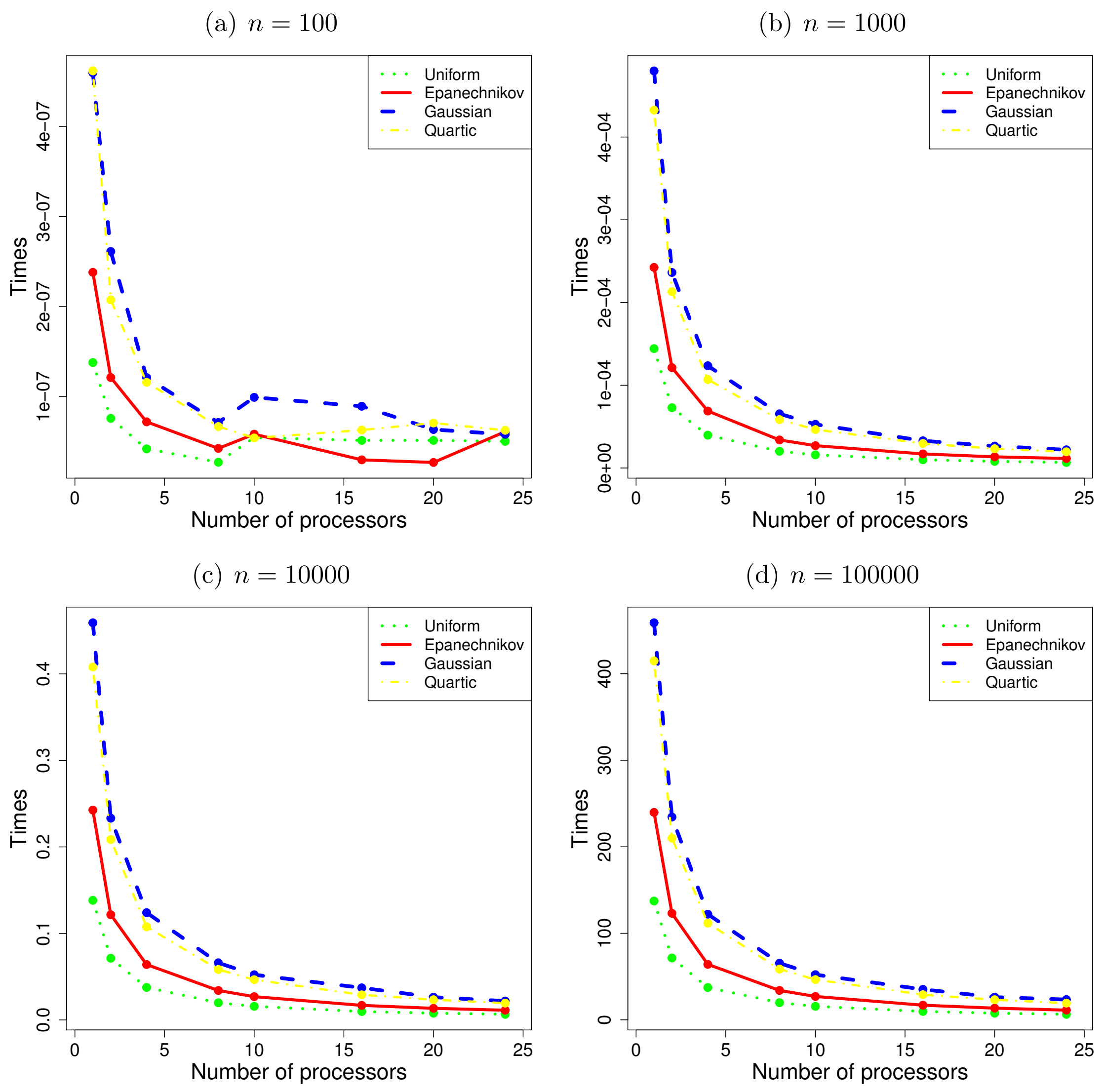


Figure 3: Benchmarks for kernel density estimation with different type of kernel functions for sample size n from 100 to 100000.

communications overhead suffered by adding more processors outweighs the potential reduction in run-time. An obvious solution is to briefly test one's application on a small sample to determine whether this is a problem. This ought only become a problem when the sample size is small relative to the number of processors in which case one may not require a parallel environment to begin with, however, one would be well-advised to conduct a simple check when dealing with moderate amonuts of data.

5. Conclusion

In this paper, we outline a parallel implementation for kernel density estimation for fixed, variable, and adaptive kernel estimators. The method is seen to achieve an almost ideal speed-up, is simple to implement, all supporting software such as compliers and the OpenMP libraries are freely available, and can be implemented in many existing multi-processors computers. We have used GNU prof and PGI prof to diagnose the execution time and have carried out the benefit of parallel implementation by the simulatoin study. It should be apparent that a parallel implementation is not only faster but can yield close to a frictionless speed-up when some care is appropriately taken.

References

Abramson, I. S. (1984). Adaptive Density Flattening–A Metric Distortion Principle for Combating Bias in Nearest Neighbor Methods. *The Annals of Statistics*, 12(3):880–886.

Breiman, L., Meisel, W., and Purcell, E. (1977). Variable Kernel Estimates of Multivariate Densities. *Technometrics*, 19(2):135–144.

Hager, G. and Wellein, G. (2011). *Introduction to high performance computing for scientists and engineers.* Chapman & Hall/CRC computational science series ; 7. CRC Press, Boca Raton, FL.

Loftsgaarden, D. O. and Quesenberry, C. P. (1965). A Nonparametric Estimate of a Multivariate Density Function. *The Annals of Mathematical Statistics*, 36(3):1049–1051.

Rosenblatt, M. (1956). Remarks on Some Nonparametric Estimates of a Density Function. *The Annals of Mathematical Statistics*, 27(3):832–837.

Stone, M. (1974). Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 111–147.