Parallel Computing of Kernel Density Estimation with Different Multi-core Programming Models

Panagiotis D. Michailidis
University of Western Macedonia
Florina, Greece
Email: pmichailidis@uowm.gr

Konstantinos G. Margaritis
University of Macedonia
Thessaloniki, Greece
Email: kmarg@uom.gr

Abstract—Kernel density estimation is nowadays very popular tool for nonparametric probabilistic density estimation. One of its most important disadvantages is computational complexity of computations needed, especially for large data sets. One way for accelerating these computations is to use the parallel computing with multi-core platforms. In this paper we parallelize two kernel estimation methods such as the univariate and multivariate kernel estimation from the field of the computational econometrics on multi-core platform using different programming frameworks such as Pthreads, OpenMP, Intel Cilk++, Intel TBB, SWARM and FastFlow. The purpose of this paper is to present an extensive quantitative (i.e., performance) and qualitative (i.e., the ease of programming effort) study of the multi-core programming frameworks for these two kernel estimation methods.

Keywords—Kernel density estimation; Parallel computing; Multi-core; parallel programming;

I. INTRODUCTION
Nonparametric methods are becoming more commonplace for applied data analysis, modeling and inference. One of its main tools is kernel density estimation (KDE) and it has been successfully applied to a large number of application domains spanning a range of fields including computational econometrics, market analysis and biostatistics to name but a few. There are numerous publications and references about the kernel density estimation methods mostly concerning theoretical and practical aspects of the estimation methods; see for example Silverman [25], Wand and Jones [27] and Klemel [19].

Kernel density estimation methods are typical of computational order $O(n^2k)$ where $n$ is the number of observations and $k$ the number of variables and in many cases the data sets are becoming larger in recent years and these kernel estimation methods are becoming more computer intensive as econometricians estimate more complicated models and utilize more sophisticated estimation techniques. Methods of data-based bandwidth selection such as cross-validation have also high computational requirements [15].

Few approximation techniques have been proposed for reducing the huge computational requirements of kernel density estimation methods. The first of them, proposed by Silverman [26], uses Fast Fourier Transform (FFT). The other one applies Fast Gauss Transform (FGT) as suggested by Elgamall [16].

An alternative way to satisfy the computational demands of kernel estimation methods is to use the parallel computing with cluster of workstations and multi-core platforms. The most important idea of parallel computing is to divide a large-scale problem into a number of smaller problems that can be solved concurrently on independent computers. There are many references about parallel computing for related non-parametric and econometric methods and applications; see for example, Adams et al [5] and Greel and Coffe [13] for a review and the monographs by Kontoghiorghes [20], [21] treats parallel algorithms for statistics and linear econometric models. However, in the field of the parallelization of kernel density methods there are a few research works. For example, Racine [23] presented a parallel implementation of kernel density estimation on a cluster of workstations using MPI library. Further, Creel [12] implemented the kernel regression method in parallel on a cluster of workstations using MPI toolbox (MPITB) for GNU Octave [17]. Recently, Lukasik [22] presented three parallelizations for kernel estimation, bandwidth selection and adaptation on a cluster of computers using MPI programming model. The parallelization of kernel estimation methods of previous papers are based on the data partitioning technique where each computer executes the same operations on different portions of a large data set.

Based on research background, there isn’t an extensive research work in the field of the parallelization of kernel estimation methods on multi-core platforms. For programming multi-core processors there are many representative parallel programming frameworks to simplify the parallelization of the computationally-intensive applications. These frameworks are Pthreads [11], OpenMP [4], Intel Cilk++ [1], Intel TBB [2], SWARM [9] and FastFlow [6], [7]. These frameworks based on a small set of extensions to the C programming language and involve a relatively simple compilation phase and potentially much more complex runtime system. We must note that there is a little related work on comparing different parallel programming frameworks on multi-core platform for several applications. For example, a
recent research work such as Kegel et al [18] that analyzes and compares three programming models such as Pthread, OpenMP and Intel TBB for parallelizing a real-world application from the area of medical imaging. Further, the work of Shekhar et al [14] compare the three popular programming models OpenMP, GCD and Pthreads to parallelize face detection and automatic speech recognition applications.

Our main contribution is to parallelize the kernel density estimation methods for univariate and multivariate data on multi-core platform using different parallel programming models. Moreover, we evaluate these two parallel kernel estimation methods both quantitatively (i.e., performance) and qualitatively (i.e., the ease of programming effort) in order to conclude which multi-core programming models are efficient for implementing these kernel methods. Moreover, this work extends from previous works such as [18], [14] and unifies the quantitative and qualitative comparison for all parallel programming frameworks for parallelizing of a time-consuming statistical application such as kernel estimation.

II. Multi-core Programming Models

This section we present a short review for all multi-core programming environments that are evaluated in this paper.

POSIX threads (in short, Pthreads) [11] is a commonly portable API (Application Programming Interface) used for programming shared memory multiprocessors and multi-core processors. This API is a low-level library. Hence, it provides the programmer a greater control about how to exploit parallelism at the expense of increasing the difficulty to use it. In the Pthreads programming tool the programmer must create all threads explicitly and use or insert all the necessary synchronization between threads. Pthreads provides a rich set of synchronization primitives such as locks, mutexes, spinlocks, Read/Write-locks, barriers and condition variables.

OpenMP [4] is a quite popular and portable API for shared memory parallel programming. Programming using OpenMP is based on the use of compiler directives which tell the compiler which parts of the code should be parallelized and how. These directives provide the programmer to create parallel sections, mark parallelizable loops and define critical sections. When a parallel loop or parallel region is defined, the programmer must specify which variables are private for each thread, shared or used in reductions. OpenMP also provides the programmer with a set of scheduling clauses to control the way the iterations of a parallel loop are assigned to threads, the static, dynamic and guided clauses. If the schedule clause is not specified, static is assumed in most implementations. Finally, the OpenMP provides some library functions to access the runtime environment. With the most recent version of OpenMP a new way of parallelization is available – called the task construct – which allows the programmer to declare and add tasks that can be executed by any thread, despite which thread that encounters the construct, i.e., an implementation of the task concept.

The Intel Cilk++ [1] language is based on technology from Cilk [10], a parallel programming model for C language. Cilk++ is an extension of the C++ language to simplify writing parallel applications that efficiently exploit multiple processors. More specifically, the Cilk++ language provides the programmer to insert keywords (cilk_spawn, cilk_sync, and cilk_for), into sequential code to tell the compiler which parts of the code that should be executed in parallel. Cilk++ also provides reducers, which eliminate contention for shared variables among tasks by automatically creating views of them for each task and reducing them back to a shared value after task completion. Moreover, the Cilk++ language is particularly well suited for, but not limited to, divide and conquer algorithms. This strategy solves problems by breaking them into sub-problems that can be solved independently, then combining the results. Recursive functions are often used for divide and conquer algorithms and are well supported by the Cilk++ language. Finally, Cilk++ provides some additional tools like performance analysis and the race condition detector Cilkscreen.

Intel Threading Building Blocks (in short, TBB) [2] is an open source library that offers a rich methodology to express parallelism in C++ programs and take advantage of multi-core processor performance. In TBB, the programmer specifies tasks of the program instead of threads and the threads are completely hidden from the programmer. The idea of TBB is to extend C++ with higher level and task-based abstractions for the parallel programming. The runtime system automatically schedules tasks onto threads in a way that makes efficient use of a multi-core platform. TBB emphasizes data parallel programming model, enabling multiple threads to work on different parts of a data collection enabling scalability to larger number of cores. Finally, TBB uses a runtime-based programming model and provides programmers with generic parallel algorithms based on a template library similar to the standard template library (STL). More specifically, TBB is based on template functions (parallel_for, parallel_reduce, etc), where the programmer specifies the range of data to be accessed, how to partition the data, the task to be executed in each chunk.

SoftWare and Algorithms for Running on multi-core (in short, SWARM) [9] is an open source parallel programming library. This library provides basic primitives for multithreaded programming. The SWARM library is a descendant of the symmetric multiprocessor (SMP) node library component of SIMPLE [8]. SWARM is built on POSIX threads that allows the programmer to use either the already developed primitives or direct thread primitives. SWARM has constructs for parallelization, restricting control of threads, allocation and deallocation of shared
memory, and communication primitives for synchronization, replication and broadcast.

FastFlow [6], [7] is an open source and C++ parallel programming framework for the development of efficient applications for multi-core computers. FastFlow is conceptually designed as a stack of layers that progressively abstract the shared memory parallelism at the level of the cores up to the definition programming constructs supported structured parallel programming on shared memory multi-core and many-core platforms. The core of the FastFlow framework is based on efficient Single-Producer-Single-Consumer (SPMC) and Multiple-Producer-Multiple-Consumer (MPMC) FIFO queues, which are implemented in a lock-free and wait-free synchronization base mechanisms. The upper level of the FastFlow framework provides a high-level programming based on parallel patterns. More specifically, FastFlow provides the programmers with a set of patterns implemented as C++ templates: farm, farm with feedback and pipeline patterns as well as their arbitrary nesting and composition. A FastFlow farm is logically built out of three entities: emitter, workers, collector. The emitter dispatches stream item to a set of workers which compute the output data. Results are then gathered by the collector back into a single stream.

III. MULTI-THREADING KERNEL DENSITY ESTIMATION

In this section we give the description of kernel density estimation methods and we also discuss how they can be parallelized using the reviewed parallel programming environments that we examined in the Section II.

A. Kernel Density Estimation

Most statistical inferences heavily depend on distribution theory (the density function). A density can give an intuitive picture of such characteristics as skewness of the distribution or the number of modes. A further advantage of having an estimate of the density is ease of interpretation for non-statisticians. In econometrics, kernel density estimation is a non-parametric way to estimate the probability density function of a random variable. Kernel density estimation is a fundamental data smoothing problem where inferences about the population are made, based on a finite data sample [3]. We begin with the simplest kernel estimator that is called a univariate density estimator. Consider a random vector \( x = [x_1, x_2, \ldots, x_n]^T \) of random variable \( x \) of length \( n \). Drawing a random sample of size \( n \) in this setting means that we have \( n \) observations of the random variable \( x \) and \( x_i \) is denoted \( i \) observation of the random variable \( x \). Our goal now is to estimate the kernel density of the random variable \( x = [x_1, x_2, \ldots, x_n]^T \) that originally proposed by Rosenblatt which is defined as [23]

\[
\hat{f}(x_i) = \frac{1}{n} \sum_{j=1}^{n} \frac{1}{h_j} K \left( \frac{x_i - x_j}{h_j} \right), i = 1, 2, \ldots n
\]

where \( K(z) \) is the kernel function that satisfies \( \int K(z)dz = 1 \) and some other regularity conditions depending on its order, and \( h_j \) is a bandwidth satisfying \( h_j \to 0 \) as \( j \to \infty \). We must note that the bandwidth which is used in formula 1 is adaptive with weights depending on \( x_i \) rather than \( x_j \).

We present the following sequential implementation in C language for a univariate adaptive bandwidth density estimator as follows:

```c
for(i = 0; i < n; i++) {
    sum_ker = 0.0;
    for(j = 0; j < n; j++) {
        sum_ker += kernel((x[i] - y[j]) / h[j]) / h[j];
    }
    pdf[i] = sum_ker / (double) n;
}
```

We must note that the kernel is a C user-defined function which contains the formula of the Gaussian kernel function.

Extension of the above approach to multivariate density estimation is straightforward so that involving \( k \) random variables. Consider a \( k \)-dimensional random vector \( x = [x_1, \ldots, x_k]^T \) where \( x_1, \ldots, x_k \) are one-dimensional random variables. Drawing a random sample of size \( n \) in this setting means that we have \( n \) observations for each of the \( k \) random variables, \( x_1, \ldots, x_k \). Suppose that we collect the \( i \)th observation of each of the \( k \) random variables in the vector \( x_i \), i.e., \( x_i = [x_{i1}, \ldots, x_{ik}]^T \) for \( i = 1, 2, \ldots, n \), where \( x_{ij} \) is the \( i \)th observation of the random variable \( x_j \). We adapt the univariate kernel density estimator to the \( k \)-dimensional case using a product kernel. Therefore, the multivariate kernel density estimator is defined as

\[
\hat{f}(x_i) = \frac{1}{n} \sum_{j=1}^{n} \prod_{d=1}^{k} \frac{1}{h_{jd}} K \left( \frac{x_{id} - x_{jd}}{h_{jd}} \right), i = 1, 2, \ldots n
\]

Now, we present the following sequential implementation in C language for a multivariate adaptive bandwidth density estimator as follows:

```c
for(i = 0; i < n; i++) {
    sum_ker = 0.0;
    for(j = 0; j < k; j++) {
        prod_ker = 1.0;
        for(k = 0; k < n; k++) {
            prod_ker *= kernel((x[i][k] - y[j][k]) / h[j][k]) / h[j][k];
        }
        sum_ker += prod_ker;
    }
    pdf[i] = sum_ker / (double) n;
}
```

We must note that in this paper we implement the univariate and multivariate adaptive bandwidth density estimator with Gaussian kernel function. These two kernel estimation methods share a common feature - they involve \( O(n^2k) \) computations in contrast to, say, the univariate density estimation that involve only \( O(n^2) \) computations (in this case, \( k = 1 \)).

B. Parallelization

The parallel processing solution to the univariate kernel estimation i.e., the formula 1 involves the partitioning of the
vector $x$ into blocks of equal size, i.e., $\lfloor n/p \rfloor$ (where $n$ is the number of elements of vector and $p$ is the number of cores) so that each core can calculate a block of $\lfloor n/p \rfloor$ independent sum kernels. Moreover, we follow similar parallel solution to the multivariate kernel estimation, i.e., the formula 2 that involves the partitioning of the matrix $x$ into blocks of rows of equal size, i.e., $\lfloor n/p \rfloor$ so that each core can calculates a block of product and sum kernels. In other words, the parallleliztion of two kernel estimation methods is based on the parallelization of the outer for loop $i$ in the above two portions of computer code as we discussed in section A. Below we present details for the parallelization of two kernel estimation methods with the six parallel programming frameworks using the proposed data partitioning. We must note that it is impossible to present the portions of code in different programming models due to space limitations.

For the implementation of two kernel estimation methods with the PThreads model, a number of threads are created using the function `pthread_create` which takes a function pointer and a pointer to that functions arguments. The call of the function `pthread_create` creates a thread that executes the function and passes the specified arguments to it (such as the rank of thread). For the parallel implementation of the two kernel methods, we created a function that embraces the original outer loop and the remaining code of the corresponding kernel method but the bounds of outer loop are modified so that each thread works on its own local block of data. The bounds of loop are based on the rank of the thread as follows: each thread works on its own block of data $b = \lfloor n/p \rfloor$ with range from $rank \times b$ to $(rank + 1) \times b$. Finally, the threads are synchronized after the computation using the `pthread_join` function.

For the implementation of the kernel methods with the OpenMP model, we introduced the `#pragma omp parallel for` directive at the beginning of the outer loop. With this combined compiler directive declares the subsequent code portion (i.e., the parallel region) to be executed by a number of parallel threads and also distributes the loop’s iterations to the threads executing the parallel region. Moreover, the distribution of loop’s iterations to threads is done by a scheduling strategy of the OpenMP. OpenMP supports different scheduling strategies specified by the `schedule` clause. In our case, we didn’t specify a strategy in the schedule clause and therefore the OpenMP model creates evenly sized blocks of loop iterations for all threads of a parallel region.

For the parallelization of kernel methods with the Cilk++ model, we replaced the outer for loop of two kernel methods by `cilk_for` keyword in order to process loops in parallel. This keyword divides the loop into blocks containing one or more loop iterations. Each block is executed serially and is spawned as a block during the execution of the loop. The maximum number of iterations in each block is the grain size. We also introduced the `cilk_grainsize` clause at the beginning of the outer loop and we specified the grain size for one loop to the value of $\lfloor n/p \rfloor$ according to our data partitioning strategy. This is all to parallelize the outer loop of the original code for two kernel estimation methods.

For the parallelization of kernel methods with the TBB model, we replaced the original outer loop of two kernel methods by a call to the `parallel_for` template function to execute loops in parallel. The `parallel_for` template function requires two arguments: the loop’s iteration space information and a function object as parameter. The iteration space is defined by three parameters such as the lower bound, the upper bound, and the optional increment of the iteration space. In our application case, the lower bound is 0 and the upper bound is $n$. On the other hand, the function object is normally declared within a separate class for each function. In other words, we created a new class for each kernel method, whose members correspond to the variables that are outside the scope of the loop to be parallelized. The class has to overload the function operator method, such that it takes an argument of type `blocked_range<T>`, which defines an iteration range. The method’s body takes the original loop or code of the corresponding kernel method; the loop’s iteration bounds are specified dynamically by the argument passed to the function operator method.

For the implementation of kernel methods with the SWARM framework, we replaced the original outer for loop by a `SWARM_pardo` construct for executing loops concurrently on one or more processing cores. This simple construct implicitly partitions the loop among the cores without the need for coordinating overheads such as synchronization of communication between the cores.

Finally, for the parallelization of kernel methods with the FastFlow model, we used the FastFlow accelerator which is a software device that can be used to speedup skeleton structured portions of code using the cores left unused by the main application. In other words, it’s a way FastFlow supports to accelerate particular computation by using a skeleton program and offloading to the skeleton program tasks to be computed. For this reason, we have written a skeleton program using the FastFlow skeletons (i.e., farm template), computing the tasks that will be given to the accelerator. This skeleton program used to program the accelerator is supposed to have an input stream, used to offload the tasks to the accelerator. Then, the skeleton program must be run using a method, such as `run_then_freeze()` method. This method will start the accelerator skeleton program, consuming the input stream items to produce either output stream items to consolidate (partial) results in memory. Finally, we created a new class task for each kernel method, whose members correspond to the variables that are outside the scope of the loop to be parallelized. The class has to overload the function task method, such that it takes an argument such as the rank of the task. The method’s body
takes the original outer loop or code of the corresponding kernel method; the loop’s iteration bounds are based on the rank of task by the argument passed to the function task method so that each task works on its own block of data.

The expertise of the programmer to implement the code of two kernel estimation methods for the different programming models is medium.

IV. RESULTS

In order to gain an insight into the practical behavior of each one of the reviewed programming model for implementing of two kernel estimation methods, we carried out a quantitative and a qualitative comparison.

A. Quantitative Comparison

For the quantitative or performance comparison we have been performed some computational experiments. The experiments were run on an Dual Opteron 6128 CPU with eight processor cores (16 cores total), a 2.0 GHz clock speed and 16 Gb of memory under Ubuntu Linux 10.04 LTS. During all experiments, this machine was in exclusive use by us. Two kernel estimation methods (i.e., univariate and multivariate) have been implemented in C/C++ programming language using all reviewed multi-core programming models. For compiling of the multi-thread kernel estimation we used three compilers. For compiling of the Pthread, OpenMP and SWARM programs we used the C compiler from the GNU Compiler Collection (GCC version 4.4.3) since it is a very widely used compiler. For compiling of the Cilk++ program we used the cilk++ which is a wrapper compiler around GCC and is the only compiler available for Cilk++. Finally, for compiling of the TBB, and FastFlow programs we used the g++ compiler version 4.4.3 which is part of the GCC collection.

Several sets of data matrices and vectors were used to evaluate the performance of the multi-thread univariate and multivariate kernel estimation, a set of randomly generated input matrices or vectors with sizes ranging from $1024 \times 1024$ to $5120 \times 5120$. Moreover, we set the parameter $k$ of two kernel estimators to a constant and maximum value of 1024 variables. To assess the performance of the multi-thread kernel estimation methods for all programming models, we used the practical execution time and speedup as a measure. The practical execution time is the total time that an multi-thread algorithm needs to complete the computation. The execution time is obtained by calling the C function `gettimeofday()` and it is measured in seconds. To decrease random variation, the execution time was measured as an average of 20 runs. On the other hand, the speedup is defined as the serial runtime of the C optimized sequential program when run on a single core processor, divided by the time of the optimized multi-thread program on the multicore processor.

Figure 1 shows the single-core performance evaluation and compiler optimizations of the six programming models for the two kernel density estimation methods. To show the effects of the compiler optimizations on the actual performance of the six programming models, we have tried different compiler optimization switches with the gcc/g++ compilers. More specifically, to present the effects of compilation on the actual performance, we compare the execution time with disable compiler optimizations (i.e., the option -O0 is used) to the one with full compiler optimizations (i.e., the option -O3 is used). From the graphs of Figure 1 we observe that the implementations with full optimization yield best performance results in relation to the implementations without optimizations. Furthermore, the performance of non-optimized versions of two kernel estimation methods are very close as the matrix size is increased. On the other hand, the performance of among the optimized implementations for the two kernel methods present some small deviations as the matrix size is increased. This phenomenon is due the fact that the some optimization techniques (i.e., vectorization, etc) of the option -O3 from GNU compiler have different effect on some implementations. Moreover, the performance gap between the implementations with full optimization and implementations without optimization widens as the matrix size is increased. Finally, we observe that the TBB and FastFlow implementations for the two kernel estimation methods benefited more from full compiler optimization than others. On the other hand, the Pthread, OpenMP and Cilk++ optimized implementations for the two kernel estimation methods benefited less from the compiler. This is due to the fact that the above programming models occur some overhead on a single core as we discussed below.

Figures 2 and 3 present the mean parallel execution times and speedups obtained for the two kernel estimation methods using all reviewed multi-core programming models, respectively. These performance results are based on the parallelization of the full optimized sequential implementations. It is also necessary to mention that the mean execution time and speedups are referred to the average value for all problem sizes (from $1024 \times 1024$ to $5120 \times 5120$) because it is impossible to present the execution times and speedups of all problem sizes due to space limitations.

Based on the graphs of Figures 2 and 3, we can say that the mean execution time of all programming models for the two kernel estimation methods is decreased significantly as the number of cores is increased. Furthermore, from figures we can see that the Pthread, OpenMP and Cilk++ incur some overhead in two kernel estimation methods on a single core but all the six multi-core implementations show identical performance when multiple cores are used. This overhead is due to the use of the `pthread_create` and `pthread_join` functions in the Pthread model for creating and terminating of a thread, the use of the `omp_for` directive in the OpenMP model which has an implicit barrier.
Figure 1. Single-core execution times (in secs) of the two kernel estimation methods as a function of the matrix size

Figure 2. Mean execution times (in secs) and speedups of the univariate kernel estimation as a function of the number of cores

Figure 3. Mean execution times (in secs) and speedups of the multivariate kernel estimation as a function of the number of cores
synchronization at the end and the use of $\text{cilk}_{\text{for}}$ construct in the Cilk++ model. The TBB, SWARM and FastFlow implementations for the two kernel estimation methods show no overhead for one core and scales best for all number of cores. We can also see that speedups are close to the 45° line for the two kernel methods with some exceptions. The speedups of the Cilk++ and FastFlow implementations for the univariate kernel method deviates from the 45° line significantly for large number of cores. The same remark is applied for the Cilk++ implementation for the multivariate kernel method. This performance degradation is due to the fact that we specified the grain size for one $\text{cilk}_{\text{for}}$ loop using the $\text{cilk}_{\text{grainsize}}$ pragma to a value of $n/p$ instead of the system calculates a default grainsize value that may works well for most loops. Furthermore, the low performance of the FastFlow implementation for large number of cores is discussed below. Finally, with a more computationally demanding method such as the multivariate kernel method with complexity $O(n^2k)$ achieves higher speedups than the univariate kernel estimation method.

From the graphs of Figures 2 and 3, we can make specific performance remarks. For the univariate kernel estimation, the TBB and SWARM have the best performance at execution time for any number of cores. On the other hand, the Cilk++ implementation has poor performance results slightly and this is due to the fact that we didn’t use the default setting of grainsize for the $\text{cilk}_{\text{for}}$ loop as discussed earlier. Furthermore, we observe a performance degradation of FastFlow implementation for large number of cores in relation to the small number of cores. This fact is due to the use of a built-in software accelerator (i.e., a collection of threads) for implementing the univariate kernel method and this accelerator provides additional synchronization mechanisms (i.e., locks and wait) in order to manage internally the tasks. This overhead of internal synchronization mechanisms is obvious for large number of cores is compared to the small amount computation of the univariate kernel method. This overhead may be eliminated by an experienced programmer because the FastFlow programming model requires medium learning curve.

Finally, for the multivariate kernel estimation we observe that the FastFlow and TBB implementations are the best for small number of cores whereas the SWARM provides satisfactory performance results for large number of cores. On the other hand, the Cilk++ give slightly poor results.

B. Qualitative Comparison

To assess the ease of programming effort in the multicore programming frameworks, we have to take into account a series of software engineering parameters such as lines of code, library popularity, level of abstraction, support for online help facilities and documentation and learning curve. As far as the lines of code, we counted the number of lines of code needed to solve the problem in each case except for the variables declaration. In Table I we present the number of lines of code for the two kernel estimation methods that required by each framework.

Based on this table, we can make the following remarks: The SWARM, OpenMP and Cilk++ implementations were identical in number of lines. The SWARM implementation allow the programmers to use constructs for parallelization, i.e., in a for loop to parallelize should be used the $\text{par}_{\text{do}}$ construct which implicitly partitions the loop among the cores without the need for coordinating overheads such as synchronization of communication between the cores. Furthermore, the SWARM framework provides the programmer library functions for synchronization and reduction operations. In the OpenMP and Cilk++ implementations, the programmer inserts compiler directives (i.e., pragma) or keywords into sequential code to tell the compiler which parts of the code that should be executed parallel. Moreover, the codes of the SWARM, OpenMP and Cilk++ frameworks are easier to understand and it isn’t required complex programming effort. Finally, as we observed in previous section, the SWARM, OpenMP and Cilk++ frameworks express the data parallelism easily because these frameworks provide ready-to-use parallel implementations without any additional lines of code. This observation is supported by the fact that the number of lines of code required from these three programming models is smaller than the other ones.

The Pthread implementations require more lines of code and the programming effort was complex. In the Pthread programming model provides the programmer low-level library routines and the parallelization of a algorithm isn’t automatic in relation to the rest frameworks, i.e., the programmer is responsible to write the code of parallelism, the distribution of data to each thread and the synchronization of the threads.

Finally, the TBB and FastFlow algorithm implementations required many lines of code although these provide C++ templates. We must note that the code which is obtained by the TBB and FastFlow programming models is easier to understand but because each algorithm is implemented as a object class is required additional statements except for the core code of the algorithm. In other words, these models require restructuring of the sequential code so that the code to be more object oriented. Therefore, these programming models require enough programming effort.

As far as the other software engineering parameters, we observe that the Pthread and OpenMP programming models are the most popular by programmers because they developed earlier and these are used in many research works by all researchers of parallel computing. The remaining programming models have medium and low popularity because of they are developed now with appearance of multicore processors. As far as the level of abstraction, the Pthread framework is low because the programmer is responsible to write code for managing of the thread parallelism and synchronization whereas the other frameworks provide high
level that can implement the same kind of thread parallelism as with Pthread model with ready-to-use parallel constructs. These frameworks except for the Pthread provide an abstract programming model, which allows the programmer to hide the details of thread programming, i.e., do not have to manage threads directly as in the Pthreads. Moreover, the Pthread, OpenMP and the two frameworks of Intel (such as Cilk++ and TBB) support online documentation/books and these provide many code examples. In particularly, the Pthread and OpenMP frameworks provide many and very good teaching tutorials. On the other hand, the SWARM and FastFlow frameworks provide a few manuals and code examples. Finally, the learning curve for the Pthread framework is long for novices programmers because it is low-level programming whereas the frameworks such as OpenMP, Cilk++ and SWARM have small learning curve because the programmers inserts compiler directives or uses ready parallelization constructs into the sequential code. The TBB and FastFlow frameworks have medium learning curve because the programmers must study lot of terms and terminology in order to understand the functionality of the object-oriented design. All above characteristics of multi-core programming models is summarized in Table II.

V. CONCLUSIONS

In this paper, we parallelized the two kernel density estimation methods from computational statistics and econometrics using all reviewed multi-core programming models. Moreover, we performed a computational quantitative and qualitative comparison of the programming models in order to answer the question which is the appropriate model for implementing the kernel methods on multi-core. Based on the performance and qualitative comparison we can conclude that the Intel TBB and SWARM programming environments are more efficient models for parallelizing the computationally-intensive computations. The reason for which these tools are efficient because they give good performance and simplicity of programming.

As future work, we could extend the quantitative and qualitative comparison of the programming models for other similar applications from the area of computational statistics and data analysis.

REFERENCES


<table>
<thead>
<tr>
<th>Statistical Method</th>
<th>Pthread</th>
<th>OpenMP</th>
<th>Cilk++</th>
<th>TBB</th>
<th>SWARM</th>
<th>FastFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Univariate Kernel Estimation</td>
<td>14</td>
<td>6</td>
<td>6</td>
<td>13</td>
<td>5</td>
<td>18</td>
</tr>
<tr>
<td>Multivariate Kernel Estimation</td>
<td>17</td>
<td>9</td>
<td>9</td>
<td>17</td>
<td>8</td>
<td>21</td>
</tr>
</tbody>
</table>

Table I Lines of code in the multi-core programming models
<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Pthread</th>
<th>OpenMP</th>
<th>Cilk++</th>
<th>TBB</th>
<th>SWARM</th>
<th>FastFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library popularity</td>
<td>High</td>
<td>High</td>
<td>Medium</td>
<td>Medium</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Level of abstraction</td>
<td>Low</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Help and documentation</td>
<td>Very good</td>
<td>Very good</td>
<td>Very good</td>
<td>Very Good</td>
<td>Good</td>
<td>Good</td>
</tr>
<tr>
<td>Learning curve</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
<td>Medium</td>
<td>Low</td>
<td>Medium</td>
</tr>
</tbody>
</table>

Table II

The characteristics of the multi-core programming models


