Efficient Multi-Core Computations in Computational Statistics and Econometrics

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*—The social researchers use computationally-intensive statistical and econometric methods for data analysis. One way for accelerating these computations is to use the parallel computing with multi-core platforms. In this paper we parallelize some representative computational kernels from statistics and econometrics on multi-core platform using the programming libraries such as Pthreads, OpenMP, Intel Cilk++, Intel TBB, Intel ArBB, SWARM and FastFlow. Specifically, these kernels are multivariate descriptive statistics (such as multivariate mean and multivariate covariance) and kernel density estimation (univariate and multivariate). The purpose of this paper is to present an extensive quantitative and qualitative study of the multi-core programming models for parallel statistical and econometric computations. Finally, based on this study we conclude that the Intel ArBB and the SWARM programming environments are more efficient for implementing statistical computations of large and small scale, respectively. The reason for which these models are efficient because they give good performance and simplicity of programming.**

***Keywords*-Statistics; Parallel computing; Multi-core; parallel programming;**

I. Introduction

Researchers, particularly in the social sciences use statistical and econometric methods for data analysis in order to understand the real world of social, economic and political phenomena. Within the social sciences, statistics and econometrics is one of the most data intensive fields, with, in many cases the data sets are becoming larger in recent years and econometrics methods are becoming more computer intensive as econometricians estimate more complicated models and utilize more sophisticated estimation techniques. One way to satisfy the increasing computational requirements is to use the parallel computing with 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.

Parallel computing in statistics and econometrics have quite a long research history. There are many papers about parallel computing for statistical and econometric computing mostly concerning specific methods and applications; see for example, Adams et al [6] and Greel and Goffe [13] for a review and the monograph by Kontoghiorghes [17] treats parallel algorithms for statistics and linear econometric models. Below we present a partial list of representative papers in the field on parallel computing in statistics and econometrics. Swann [20] presented a Message Passing Inferface (MPI) parallel implementation of maximum likelihood estimation for a simple econometric problem with Fortran code. We must note that the maximum likelihood method requires much computation and can be parallelized. Racine [18] presented a parallel implementation of kernel density estimation on a cluster of workstations using MPI library. Doornik et al [15] provided an example for parallel Monte Carlo simulation of the univariate normality and trace tests using an MPI subset for the Ox language. Creel [12] implemented several econometric methods such as Monte Carlo simulation, bootstrapping, estimation by maximum likelihood and kernel regression in parallel on a cluster of workstations using MPI toolbox (MPITB) for GNU Octave [16]. Sevcikova [19] presented a simple framework for parallel programming statistical simulations (i.e., Monte Carlo or bootstrapping) in PVM library. These simulation studies are used for assessing the properties of statistical tests and estimators. Rose et al [14] proposed three parallel implementations of the Bootstrap simulation technique applied in the context of Markovian simulation on multi-core symmetric multiprocessing (SMP) clusters. The first implementation is a pure MPI model and other two ones are based on hybrid programming model with MPI and OpenMP. The parallelization of econometric 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 econometric methods on multi-core platforms. For programming multi-core processors there are many representative parallel programming models to simplify the parallelization of the computationally-intensive applications. These models are Pthreads [11], OpenMP [5], Intel Cilk++ [2], Intel TBB [3], Intel ArBB [1], SWARM [9] and FastFlow [7]. These models 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.

Our main contribution is to parallelize some statistical and econometrics kernels like multivariate descriptive statistics (multivariate mean and multivariate covariance) and kernel density estimation (univariate and multivariate) on multi-core platform using these programming libraries. Moreover, we evaluate these kernels both quantitatively (i.e., performance) and qualitatively (i.e., the ease of programming effort) in order to conclude which multi-core programming libraries are efficient for implementing these econometric kernels.

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 model 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 [5] 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 parallelize 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, 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++ [2] 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) [3] 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.

Intel Array Building Blocks (in short, ArBB) [1] is an open source high-level API, backed by a library that support data parallel programming solution designed to effectively utilize the power of existing and upcoming throughput-oriented features on modern processor architectures, including multi-core and many-core platforms. Intel ArBB extends C++ for complex data parallelism including irregular and sparse matrices and works with tools such as standards C++ compilers. Therefore, ArBB is best suited for compute-intensive, data parallel applications (often involving vector and matrix math). In ArBB allow the programmers to express parallel computations with sequential semantics by expressing operations at the aggregate data collection level. For this reason, ArBB model provides a rich set of data types for representing your data collections (aggregations of data such as matrices and arryas). Fundamental elements of these data types are dense and nested containers, which are collections to which data parallel operators may be applied. All vectorization and threading is managed internally by ArBB. Furthermore, the programmer uses collective operations with clear semantics such as `add_reduce` that computes the sum of the elements in a given array. ArBB also has language constructions for control flow, conditionals and loops. These operations have their usual sequential semantics and are not parallelized by the system, rather, only specific collective operations are executed in parallel.

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 [7] is a 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 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 Statistical Kernels

In this section we give the description of kernels from computational statistics and econometrics and we also discuss how they can be parallelized using the reviewed parallel programming environments that we examined in the Section II.

A. Multivariate Descriptive Measures

The statisticians analyse data that from their nature is multivariate data. A multivariate data set is a collection of data, usually organized in tabular form. Each column represents a particular variable and each row corresponds to a given member of the data set in question. It lists values for each of the variables. The data set may comprise data for one or more members, corresponding to the number of rows. This data set usually stored in a data matrix. In general the data matrix on n observations (individuals) and p variables is written as $X = (x_{ij})_{n\times p}$, where x_{ij} is a coding of information of ith individual on jth variable. For the summarize of the data set we introduce two popular multivariate descriptive measures such as multivariate mean and multivariate covariance or correlation. The computation of these measures is a necessary preliminary to subsequence multivariate analyses in addition to being useful in its own right as a way to summarize aspects of variation in the data.

The multivariate mean is a vector of mean values where each mean value is concerned for each variable. The mean vector is usually a column vector, i.e., $\overline{x} = [\overline{x_1}\,\overline{x_2}\ldots\overline{x_p}]^T$, where

$$\overline{x_j} = \frac{1}{n}\sum_{i=1}^{n} x_{ij} \tag{1}$$

is the average of jth variable, $j = 1, 2, \ldots, p$. This computation can be reformulated in matrix algebra terms and is expressed in matrix notation as

$$\overline{x} = \frac{1}{n} X^T_{p\times n} \cdot 1_{n\times 1} \tag{2}$$

where 1 is an $n \times 1$ vector of unity and X^T is the transpose of X.

The multivariate covariance is a covariance matrix $S = [s_{jk}]_{p\times p}$, where s_{jk} is the covariance between jth and kth variables. This is calculated (based on Pearson's method) as

$$s_{jk} = \frac{1}{n}\sum_{i=1}^{n}(x_{ij} - \overline{x_j})(x_{ik} - \overline{x_k}), j, k = 1, 2, \ldots, p \tag{3}$$

where $\overline{x_j}$ and $\overline{x_k}$ are pre-computed mean vectors. On the other hand, the multivariate correlation is a correlation matrix $R = [r_{jk}]_{p\times p}$, where r_{jk} is the correlation between jth and kth variables. This is calculated as

$$r_{jk} = \frac{s_{jk}}{\sqrt{s_{jj}} \cdot \sqrt{s_{kk}}}, j, k = 1, 2, \ldots, p \tag{4}$$

The computation of covariance and correlation can be reformulated in matrix algebra terms. More specifically, the variance can be expressed in matrix notation as

$$S = \frac{1}{n-1} X^T_{p\times n} \cdot X_{n\times p} - \overline{x}_{p\times 1} \cdot \overline{x}^T_{1\times p} \tag{5}$$

where X^T is the transpose of X and $\overline{x}$ is pre-computed mean vector. The correlation can be expressed in matrix notation as

$$R = D^{-1}_{p\times p} \cdot S_{p\times p} \cdot D^{-1}_{p\times p} \tag{6}$$

where D is a diagonal matrix $D = diag(s_1, \ldots, s_p)$ and D^{-1} is the inversion matrix of D, i.e., $D^{-1} = diag(\frac{1}{s_1}, \ldots, \frac{1}{s_p})$.

For the parallelization of multivariate mean and multivariate covariance/correlation, we parallelize the computational formulas 2, 5 and 6. The formula 2 consists of two matrix operations such as matrix transpose and matrix

- vector product whereas the formulas 5 and 6 consists of five matrix operations such as matrix transpose, matrix product, matrix - vector product, outer product and matrix substration. Therefore, we parallelize these matrix operations as individual routines using the multi-core programming models of Section II. The parallel processing of these matrix operations is based on a simple data partitioning technique that involves partitioning the data such that each thread works concurrently on an local part of the data. More specifically, we divide the matrix and vector into blocks of rows of equal size, i.e., $\lceil m/c \rceil$ where m is the number of rows of any matrix or vector and c is the number of cores.

B. Kernel Density Estimation

Most statistical inferences heavily depend on 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 [4]. 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_j is denoted j 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 [18]

$$\hat{f}(x_j) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{h_i}K(\frac{x_j - x_i}{h_i}), j = 1, 2, \ldots n \qquad (7)$$

where $K(z)$ is the kernel function that satisfies $\int K(z)\mathrm{d}z = 1$ and some other regularity conditions depending on its order, and h_i is a bandwidth satisfying $h_i \to 0$ as $i \to \infty$. We must note that the bandwidth which is used in formula 7 is adaptive with weights depending on x_i rather than x_j.

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 jth observation of each of the k random variables in the vector x_j, i.e., $x_j = [x_{j1}, \ldots x_{jk}]^T$ for $j = 1, 2, \ldots n$, where x_{ji} is the jth observation of the random variable x_i. 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

$$\hat{f}(x_j) = \frac{1}{n}\sum_{i=1}^{n}\{\prod_{d=1}^{k}\frac{1}{h_{id}}K(\frac{x_{jd} - x_{id}}{h_{id}})\}, j = 1, 2, \ldots n \qquad (8)$$

We must note that in this paper we implement the univariate and multivariate adaptive bandwidth density estimator with Gaussian kernel function. Two kernels of kernel estimation involve $O(n^2k)$ computations in contrast to, say, the univariate density estimation that involve only $O(n^2)$ computations (in this case, $k = 1$).

The parallel processing solution to the univariate kernel estimation i,.e. the formula 7 involves the partitioning of the vector x into blocks of equal size, i.e., $\lceil n/c \rceil$ (where n is the number of elements of vector and c is the number of cores) so that each core can calculates a block of $\lceil n/c \rceil$ sum kernels. Finally, we follow similar parallel solution to the multivariate kernel estimation, i.e., the formula 8 that involves the partitioning of the matrix x into blocks of rows of equal size, i.e., $\lceil n/c \rceil$ so that each core can calculates a block of product and sum kernels.

IV. Results

In order to gain an insight into the practical behavior of each one of the reviewed programming model for implementing statistical kernels, 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 not performing other heavy tasks (or processes). All statistical kernels have been implemented in C/C++ programming language using all reviewed multi-core programming models. For compiling of the multi-thread statistical kernels we used three compilers. For compiling of the Pthread, OpenMP and SWARM programs we used the C compiler from the GNU Compiler Collection (GCC) since it is a very widely used compiler. For compiling of the Cilk++ program we used the Intel Cilk++ which is a wrapper compiler around GCC and is the only compiler available for Cilk++. Finally, for compiling of the TBB, ArBB and FastFlow programs we used the g++ compiler which is part of the GCC collection. It is necessary to mention that the compilation of the programs has been made without the optimization.

Several sets of data matrices and vectors were used to evaluate the performance of the multi-thread multivariate mean and correlation and the univariate and multivariate kernel estimation, a set of randomly generated input matrices or vectors with sizes ranging from 1024×1024 to

5120×5120. Moreover, we set the parameters p and k of two multivariate descriptive measures and of two kernel estimators to a constant and maximum value of 1024 variables, respectively. To assess the performance of the multi-thread statistical kernels for all programming models, we used the practical execution time and model's performance as a measures. 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 40 runs. On the other hand, model's performance for each statistical kernel has been calculated using the relation

$$\frac{BT(model)t}{T(model)i} \times 100, i = 1, 2, \ldots, 7 \qquad (9)$$

where $BT(model)t$ is the best execution time for a specific statistical kernel, run by all seven models and $T(model)i$ is the execution time of the model i for the same statistical kernel. The best timing is then set equal to 100%. To calculate the overall performance for each model, for any combination of the problem size and number of cores, we add the percentage values for every statistical kernel and divide it by the total number of kernels, i.e., 4. Finally, the largest percentage corresponds to the best overall performance.

Following in this section, there are some figures extracted from the performance evaluation experimental data. They are presented as three figures, one presents the graphs for all statistical kernels while the other two show average and overall performance. The time in the y-axis in the figures below is in logarithmic scale. Figure 1 presents the mean execution times for all the statistical kernels using all reviewed multi-core programming models. It is necessary to mention that the mean execution time is referred to the average time for all problem sizes (from 1024×1024 to 5120×5120) because it is impossible to present the execution times of all problem sizes due to space limitations.

Based on the graphs of Figure 1, we can say that the mean execution time of all reviewed models for all statistical kernels is decreased as the number of cores is increased with some exceptions. More specifically, the mean execution time of the multivariate covariance/correlation and the two kernels of Kernel estimation is decreased significantly is compared to the execution time of the multivariate mean. On the other hand, the marginal reduction in computing times of the multivariate mean dissipates as additional cores are added (i.e., one obtains a larger reduction when going from two to eight cores than when going from 8-16 cores). This is due to the fact that the multivariate mean require small number of steps i.e., n steps approximately is compared to the other statistical kernels which require n^2 steps. Another reason for low performance of the multivariate mean is that for implementing this kernel uses the matrix transpose operation. It is known that this matrix operation has poor spatial locality because scans the matrix column by column instead of row by row and it leads to occur higher cache miss rate. We must note that this matrix transpose operation is used in the multivariate covariance kernel but doesn't seem to take effect in their performance because the time of matrix transpose is very small is compared to the time of the matrix product. Finally, it is necessary to mention that for the Intel ArBB programming model we haven't seen a significant decrease at execution time as a function of the number of cores, but we have a significant decrease execution time compared to the execution time of the C sequential program. In other words, the relative speedups of ArBB implementation over the C sequential implementation for all kernels are significant (except for multivariate mean). More specifically, the relative speedup of the multivariate variance/correlation over the serial implementation ranging from 45 to 138 times faster, the speedup of the univariate kernel estimation ranging from 2 to 8 times faster and the speedup of the multivariate kernel estimation ranging from 2 to 125 times faster. These important speedups of the ArBB implementation are due to the vectorization of the model, i.e., many operations are performed at the aggregate data collection level using dense containers as a data structures. However, the relative speedup of the multivariate mean over the C serial implementation is not significant because the runtime overhead of the ArBB model dominates in relation to the low volume of computation.

From the graphs of Figure 1, we can make specific performance remarks. For the multivariate mean kernel, the SWARM implementation has the best performance at execution time for any number of cores whereas the ArBB implementation has the slowest performance. We must note that the TBB implementation gives good performance for two cores and it also presents performance closest to the SWARM implementation. The good performance of the SWARM implementation is due to the fact that there is a small runtime system overhead and therefore it works well on a problem of small and medium scale. The low performance of the ArBB implementation is explained earlier. Moreover, the performance of the remaining models are closest to the Pthread implementation.

For the multivariate covariance/correlation kernel, the ArBB and the FastFlow have the first and second best performance at execution time for any number of cores, respectively. The very good performance of the ArBB implementation is due to the vectorization and the fact that the problem is large scale. On the other hand, OpenMP implementation has the worst execution time for any number of cores and it is due to the fact that this model does not support backstage code optimization routines. The performance of the other programming models are close to OpenMP implementation with some exceptions for large number of

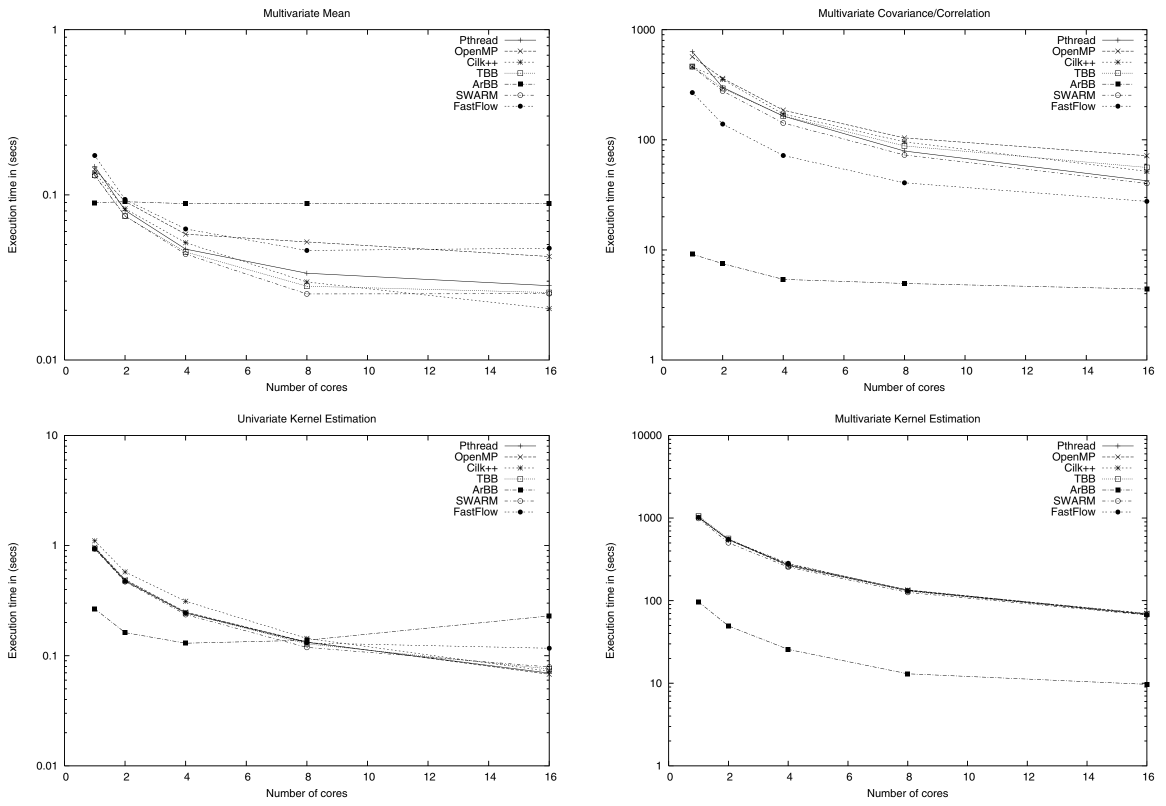

Figure 1. Mean execution times (in secs) of the statistical kernels as a function of the number of cores

cores.

For the univariate kernel estimation, the ArBB and SWARM/OpenMP have the best performance at execution time for small and large number of cores, respectively. The low performance of ArBB implementation for large number of cores is due to the fact that the runtime system overhead dominates in relation to the low volume of computation. For the rest programming models, their performance are approximately same for any number of cores.

Finally, for the multivariate kernel estimation we observe that the ArBB implementation is the winner of this comparison for any number of cores. The satisfactory performance of the ArBB implementation is due to the fact that it performs very well the vectorization on a large-scale problem. The performance of the rest implementations are identical.

In Figures 2 and 3, we present the overall performance for all the reviewed programming models as a function of the number of cores and problem sizes, respectively. We should mention that the average performance of Figure 2 for each model and number of core is the average value for all the statistical kernels and problem sizes. Similarly, the average performance of Figure 3 for each model and problem size is the average value for all the kernels and number of cores.

Based on these results, we can observe that the ArBB has the best overall performance for any number of cores and problem sizes. Moreover, the overall performance of the ArBB is decreased as the number of cores is added whereas its performance is constant as the problem size is increased. Furthermore, the next best performance in the final order is the SWARM, TBB, OpenMP, Pthread and Cilk++ programming models with very large difference from the ArBB model for any number of cores and problem sizes. Finally, at the third place comes the FastFlow model that have the lowest performance.

B. Qualitative Comparison

To assess the ease of programming effort in the multicore programming models, we have to take into account a series of software engineering parameters such as lines of code, library popularity, 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 declarations. In Table I we present the number of lines of code for all statistical kernels that required by each model.

Based on this table, we can make the following remarks: The ArBB algorithm implementations were, in general, shorter in length, more concise and hence easier to understand. The ArBB code for implementing statistical kernels is more concise because the statements or operations in

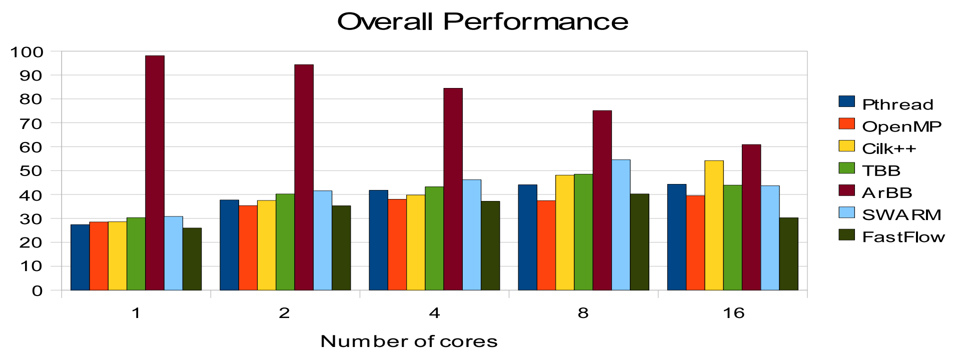

Figure 2. Overall performance as a function of the number of cores

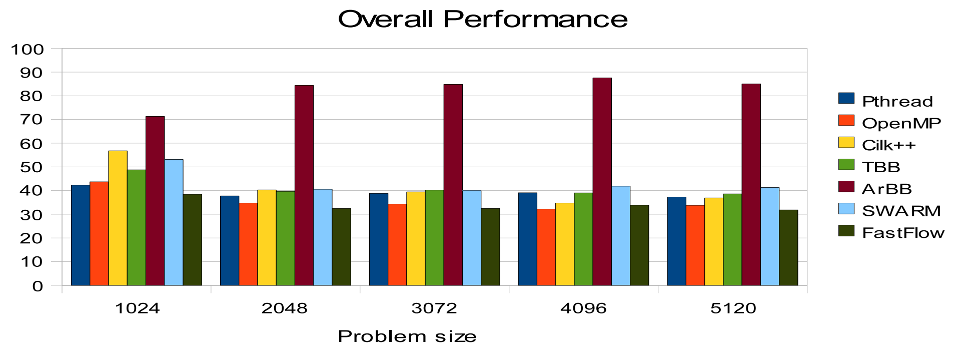

Figure 3. Overall performance as a function of the problem size

Statistical Kernel	Pthread	OpenMP	Cilk++	TBB	ArBB	SWARM	FastFlow
Multivariate mean	27	12	12	26	3	10	42
Multivariate correlation	50	26	26	53	10	21	82
Univariate Kernel Estimation	14	6	6	13	4	5	18
Multivariate Kernel Estimation	17	9	9	17	5	8	21

Table I
LINES OF CODE IN THE MULTI-CORE PROGRAMMING MODELS

the ArBB implementation are vector or expressed at the aggregate data collection level using dense containers as a data types without the use of for loops.

The SWARM, OpenMP and Cilk++ implementations were identical in number of lines and their codes were not significantly longer than ArBB implementations. The SWARM implementation allow the programmers to use constructs for parallelization, i.e., in a for loop to parallelize should be used the `par_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 model 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++ models are easier to understand and it isn't required complex programming effort.

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 models, i.e., the programmer is responsible to write the code of parallelism and the distribution of data to each thread.

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.

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 libraries have medium and low popularity because of they are developed now with appearance of multicore processors. Moreover, the Pthread, OpenMP and the three libraries of Intel (such as Cilk++, TBB and ArBB) support online documentation/books and these provide many code examples. In particularly, the Pthread and OpenMP models provide many and very good teaching tutorials. On the other hand, the SWARM and FastFlow models provide a few manuals and code examples. Finally, the learning curve for the Pthread model is long for novices programmers because it is low-level programming whereas the models such as OpenMP, Cilk++, ArBB 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 libraries 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.

V. Conclusions

In this paper, we parallelized the four representative kernels 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 statistical kernels on multi-core. Based on the performance and qualitative comparison we can conclude that the Intel ArBB programming environment is more efficient model for parallelizing the computationally-intensive statistical computations such as the multivariate covariance/correlation and the two kernels of kernel estimation such as univariate and multivariate whereas the SWARM library is efficient model for small-scale computations like the multivariate mean kernel. The reason for which these models are efficient because they give good performance and simplicity of programming.

References

[1] Intel Array Building Blocks, 2012. http://software.intel.com/en-us/articles/intel-array-building-blocks/.

[2] Intel Cilk Plus, 2012. http://software.intel.com/en-us/articles/intel-cilk-plus/.

[3] Intel Threading Building Blocks, 2012. http://threadingbuildingblocks.org/.

[4] Kernel density estimation, 2012. http://en.wikipedia.org/wiki/Kernel_density_estimation.

[5] The OpenMP API specification for parallel programming, 2012. http://openmp.org/wp/.

[6] N. M. Adams, S. P. J. Kirby, P. Harris, and D. B. Clegg. A review of parallel processing for statistical computation. *Statistics and Computing*, 6:37–49, 1996.

[7] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. *Programming Multi-core and Many-core Computing Systems*, chapter FastFlow: high-level and efficient streaming on multicore. Wiley, 2011.

[8] D. A. Bader and J. JaJa. Simple: A methodology for programming high performance algorithms on clusters of symmetric multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 58:92–108, 1999.

[9] D. A. Bader, V. Kanade, and K. Madduri. SWARM: A Parallel Programming Framework for Multicore Processors. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007*, pages 1–8, 2007.

[10] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[11] D. Buttlar, J. Farrell, and B. Nichols. *PThreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly Media, 1996.

[12] M. Creel. User-friendly parallel computations with econometric examples. *Comput. Econ.*, 26(2):107–128, Oct. 2005.

[13] M. Creel and W. L. Goffe. Multi-core CPUs, Clusters, and Grid Computing: A Tutorial. *Comput. Econ.*, 32(4):353–382, Nov. 2008.

[14] C. A. F. de Rose, P. Fernandes, A. M. de Lima, A. Sales, and T. Webber. Exploiting multi-core architectures in clusters for enhancing the performance of the parallel bootstrap simulation algorithm. In *IPDPS Workshops'11*, pages 1442–1451, 2011.

[15] J. A. Doornik, N. Shephard, and D. F. Hendry. Parallel computation in econometrics: A simplified approach. Economics Papers 2004-W16, Economics Group, Nuffield College, University of Oxford, Jan. 2004.

[16] J. Fernandez, M. Anguita, S. Mota, A. Caas, E. Ortigosa, and F. Rojas. MPI toolbox for Octave. In *VecPar'2004*, 2004. http://atc.ugr.es/~javier/investigacion/papers/VecPar04.pdf.

[17] E. J. Kontoghiorghes. *Handbook of Parallel Computing and Statistics*. Chapman & Hall/CRC, 2005.

[18] J. Racine. Parallel distributed kernel estimation. *Comput. Stat. Data Anal.*, 40(2):293–302, Aug. 2002.

[19] H. Sevcikova. Statistical simulations on parallel computers. *Journal of Computational and Graphical Statistics*, 13(4):886–906, 2004.

[20] C. A. Swann. Maximum likelihood estimation using parallel computing: An introduction to MPI. *Comput. Econ.*, 19(2):145–178, Apr. 2002.