Scientific computations on multi-core systems using different programming frameworks

Panagiotis D. Michailidis[a], Konstantinos G. Margaritis[b]

[a]*Department of Balkan, Slavic and Oriental Studies, University of Macedonia*
156 Egnatia Str., 54636 Thessaloniki, Greece
E-mail: pmichailidis@uom.gr
[b]*Department of Applied Informatics, University of Macedonia*
156 Egnatia Str., 54636 Thessaloniki, Greece
E-mail: kmarg@uom.gr

Abstract

Numerical linear algebra is one of the most important forms of scientific computation. The basic computations in numerical linear algebra are matrix computations and linear systems solution. These computations are used as kernels in many computational problems. This study demonstrates the parallelisation of these scientific computations using multi-core programming frameworks. Specifically, the frameworks examined here are Pthreads, OpenMP, Intel Cilk Plus, Intel TBB, SWARM, and FastFlow. A unified and exploratory performance evaluation and a qualitative study of these frameworks are also presented for parallel scientific computations with several parameters. The OpenMP and SWARM models produce good results running in parallel with compiler optimisation when implementing matrix operations at large and medium scales, whereas the remaining models do not perform as well for some matrix operations. The qualitative results show that the OpenMP, Cilk Plus, TBB, and SWARM frameworks require minimal programming effort, whereas the other models require advanced programming skills and experience. Finally, based on an extended study, general conclusions regarding the programming models and matrix operations for some parameters were obtained.

Keywords: Scientific Computations, Linear Algebra, Parallel Computing, Multi-core, Parallel Programming

1. Introduction

Scientific computing is a collection of quantitative methods and tools used to develop and solve mathematical models of a variety of scientific problems using a computer system [66]. Numerical linear algebra is one of the most important quantitative methods for scientific computing. The basic computations of numerical linear algebra are matrix computations (such as vector and matrix addition, dot product, outer product, matrix transpose, matrix-vector product, and matrix product) and solutions of linear systems (such as the direct Gaussian elimination method and the iterative Jacobi method), which are used as kernels in many computational problems such as computational statistics [34] and combinatorial optimisation [41, 68, 70]. To satisfy the heavy computational requirements of these methods, large-scale matrix computations use towers of software infrastructure such as BLAS/LAPACK [14], ScaLAPACK [20], and PLAPACK [69] running on cluster computing platforms. Recently, modern high-performance computer systems have been introduced, which can be classified into three categories. The first, multi-core platforms, integrate a few cores (from two to ten) on the same integrated circuit chip (die) in an effort to speed up execution of computationally intensive methods. The second, many-core platforms or general-purpose graphics processing units (GPUs), which consist of a large number of cores (as many as several hundred), are specifically oriented to maximizing execution throughput for parallel applications [29]. The third, reconfigurable platforms based on field-programmable gate arrays, are becoming important, especially when higher performance/power computation ratios are desired. Along with the introduction of these platforms, software projects such as Parallel Linear Algebra for Multi-core Architectures (PLASMA) [9] have been developed for multi-core machines and Matrix Algebra on GPU and Multicore Architectures (MAGMA) [9] for GPU platforms. Recent comparisons of these platforms have shown substantial architectural and performance differences for several application areas [18, 36, 40, 45, 71]. Therefore, this paper focusses on the

multi-core approach because it is the easiest way to speed up an application from the perspective of non-specialised developer. However, sequential implementation of linear algebra computations on a multi-core architecture will not improve performance because it cannot exploit the other cores that are available. Successful implementation of scientific computations on a multi-core machine can be achieved only through parallel (or multi-core) programming.

The fundamental programming question on a multi-core platform is how to decompose a problem into several sub-problems and how to map these to cores with the goal of increasing performance. Moreover, programmers need to study and understand the hardware characteristics of the multi-core platform to write efficient parallel programs. For this reason, programming on multi-core processors is a more complex procedure than programming on sequential processors because application data in memory can be accessed by several entities called threads, which belong to the same program. For this reason some synchronisation between threads is necessary. Therefore, there is a need to bridge the gap between hardware and software applications to hide the hardware details from the programmers and enable them to write parallel programs with minimal programming effort. This has resulted in the introduction of several parallel programming models [29, 43] which simplify the parallelisation of linear algebra computations and other related applications on multi-core computers. However, these models differ significantly in their parallel design principles, abstraction levels, semantics, and syntax. Some popular models are POSIX threads (Pthreads for short) [25], OpenMP [3], Intel Cilk Plus [1], Intel Threading Building Blocks (TBB for short) [2], SoftWare and Algorithms for Running on Multi-core (SWARM for short) [16], and FastFlow [10, 11]. These models are based on a small set of extensions to the C programming language and involve a relatively simple compilation phase and a potentially much more complex runtime system.

Based on the various features and qualities of programming models, the question posed by programmers becomes: what is the appropriate parallel programming framework for implementing linear algebra computations on a multi-core system to achieve a balance between high programmer productivity (i.e., minimal programming effort) and high performance? Therefore, the contribution of this paper is a unified and systematic quantitative (i.e., performance-based) and qualitative (i.e., related to the ease of programming effort) comparison of all multi-core programming frameworks for implementing linear algebra computations based on simple parallelisation techniques. Finally, the authors believe that this work is important and interesting because this comparison may turn out to be very helpful for other programmers and scientists who are often faced with a variety of options for implementing projects.

The rest of the paper is organised as follows. In Section 2, related work is discussed. In Section 3, an abstract multi-core system architecture is presented along with all the reviewed parallel programming frameworks, and in Section 4, parallelisation issues in implementing linear algebra computations are considered. In Section 5, a performance and qualitative evaluation of the reviewed parallel programming models for parallelising linear algebra operations is described. Finally, Section 6 presents conclusions.

2. Related work

In the research literature, several studies have evaluated various parallel programming models on multi-core platforms. Most of this research has compared parallel programming models from the performance point of view. However, there has been little related work on comparing parallel programming models from the qualitative or productivity points of view.

The research studies based on performance evaluation of different multi-core programming models can be organised into four groups:

1. *Evaluation of a programming model to parallelise a specific problem.* Research studies [51] and [53] evaluated the parallelisation performance of the Gaussian elimination and LU factorisation algorithms using the OpenMP programming model. The parallelisation of Gaussian elimination [51] was based on the data parallel approach, whereas the parallel implementation of the LU decomposition [53] was based on the pipeline approach. Zuckerman *et al.* [72] evaluated the performance of the parallel matrix multiplication kernel using a high-performance M:N threading library, Microthread, and showed its efficiency with regard to the well-known Intel Math Kernel Library (IMKL). Runger *et al.* [63] presented a parallel optimised library for dense matrix multiplication on a multi-core platform. This implementation was based on a recursive approach and was compared with efficient libraries such as GotoBLAS, IMKL, and AMD Core Math Library (AMCL). Finally, Michailidis *et al.* [55] studied the performance of the pipeline approach in the OpenMP model for parallelisation of the Gauss-Jordan

algorithm for solving systems of linear equations. The performance results were compared to the performance of two other naive parallel approaches, row block and row cyclic distribution.

2. *Evaluation of a programming model to parallelise a set of problems.* Michailidis *et al.* [54] presented performance results for the OpenMP model in the parallelisation of two important matrix computations, matrix-vector product and matrix multiplication. These parallelisations were based on a simple parallel data technique. Buttari *et al.* [23, 24] developed the PLASMA library for implementing certain linear algebra operations on a multi-core platform using the title algorithms.
3. *Evaluation of a few programming models to parallelise a specific problem.* Li [47] evaluated the performance of a high-performance sparse LU factorisation algorithm on several representative multi-core machines using the Pthreads and MPI programming models. Aldinucci *et al.* [12] demonstrated a parallel implementation of the Smith-Waterman bio-informatics algorithm using Cilk, OpenMP, Intel TBB, and FastFlow. They found experimentally that the FastFlow model performed better than the other three models. Finally, Kegel *et al.* [44] analysed and compared three programming models, Pthreads, OpenMP, and Intel TBB, for parallelising a real-world medical imaging application.
4. *Evaluation of a few programming models to parallelise a small set of problems.* Marowka [49] presented a comparative study of OpenMP and TBB micro-benchmarks to study parallelisation overheads on a dual-core machine with two different compilers, the Intel compiler and Microsoft Visual Studio C++. The design of the TBB micro-benchmark followed the design methodology of the OpenMP EPCC micro-benchmarks [22]. Ayguade *et al.* [15] analysed the performance of six algorithms, including the sparseLU and Strassen algorithms, using OpenMP, Intel task queues, and Intel Cilk programming models. Kurzak *et al.* [46] implemented matrix factorisation algorithms such as Cholesky factorisation, QR factorisation, and LU factorisation on multi-core processors using multi-thread programming tools such as Cilk and SMPSs. These implementations were based on tiling algorithms. Podobas *et al.* [59] presented a quantitative study comparing three task-based parallel programming models, OpenMP, Intel Cilk Plus, and Wool [31], for a small set of micro-benchmarks including the SparseLU and Strassen algorithms. This performance evaluation took into consideration the cost of creating and joining tasks. Ravela *et al.* [61] studied the performance of the Pthreads, OpenMP, Intel Cilk Plus, and Intel TBB programming models for a set of scientific computing benchmarks such as matrix multiplication, Jacobi iteration, and Laplace heat distribution. Furthermore, this study considered the development time required to parallelise these benchmarks. Sanchez *et al.* [65] undertook a comparative evaluation of the OpenMP, Intel TBB, Intel ArBB, and CUDA models for parallel implementations of four problems (including matrix multiplication) on five shared memory platforms. Note that the Intel ArBB model combines multi-core parallel threads and SIMD features with a simpler programming model, whereas the CUDA model exploits the SIMD features of the GPU hardware. Finally, Ali *et al.* [13] presented a performance study of OpenMP, Intel TBB, and OpenCL for five benchmark applications (including matrix multiplication and LU decomposition) on a CPU multi-core platform. This study took into consideration the compiler optimisation, overhead, and scalability of these frameworks. The OpenCL framework is a standard API (Application Programming Interface), which focusses on programming for multi-core processors except for GPU hardware.

Note that other research papers have compared various parallel programming frameworks on a multi-core platform for several applications. For example, [52] compared OpenMP, Cilk Plus, Intel TBB, and OpenCL for implementing a 2D/3D image registration algorithm. Furthermore, Shekhar *et al.* [27] compared three popular programming models (OpenMP, GCD, and Pthreads) for parallelising face detection and automatic speech recognition. However, few studies have focussed on a qualitative comparison of parallel programming models. Earlier works such as [42, 67] evaluated programming effort and usability for two parallel programming models, OpenMP and MPI. In addition, Ali *et al.* [13] presented a qualitative comparison of three models, OpenMP, Intel TBB, and OpenCL, for a set of applications including matrix multiplication. This comparison was based on two software engineering criteria: time in weeks for learning and mapping each application into the corresponding framework by the user, and the number of lines of code for each parallelised application using these models. Finally, Kegel *et al.* [44] analysed the number of lines of code for a parallelised medical imaging application using the Pthreads, OpenMP, and TBB models.

Other studies [56, 57, 58] have presented a preliminary experimental and qualitative study of programming models and matrix operations for a limited set of performance parameters. These studies looked at performance as a function of the number of cores without considering several compiler optimisation options or problem scalability, whereas the

qualitative study was limited to an analysis of the number of lines of code required to parallelise matrix operations.

Type of comparison - Method	1:1	1:F	F:1	F:F
Performance evaluation	+++	+++	+++	+++
Qualitative evaluation	- - -	- - -	- - -	+
Performance and Qualitative eval.	- - -	- - -	+	+

Table 1: Summary of related work for given different evaluation techniques of multi-core programming models for three types of comparison [Note: left of the (:) means one (1) or few (F) programming models and right means one (1) or few (F) problems, Notation of scale: +++ means relatively high research activity, - - - means relatively low research activity]

Table 1 compares the research studies for the four different evaluation techniques for multi-core programming models and considers whether each is a performance evaluation, a qualitative evaluation, or both. From the table, it is clear that no systematic and unified study or comparison can be made from the quantitative or qualitative point of view. Furthermore, these research studies compared only a limited number programming models, i.e., two or three, for a limited set of scientific computing problems. Moreover, most studies were either quantitative or qualitative, although a few have combined these two types of comparison, but at limited depth. For this reason, the present paper differs from previous research because it presents a unified and extensive quantitative and qualitative comparison of a larger number of multi-core programming models for a larger set of linear algebra applications. The quantitative portion of this work considers performance with compiler optimisation and problem scalability. Furthermore, the qualitative comparison of models in this paper extends to other software engineering criteria, except for the number of lines of code, as discussed later.

3. Multi-core architecture and programming models

This section presents an abstract computational model for a multi-core architecture and a short review of the multi-core programming models evaluated in this paper.

A typical multi-core system ([50], p. 45) consists of a finite number of identical processing cores integrated on a single chip. Each processing core has its own private L1 cache memory and shares the L2 cache memory with other cores. In such a design, the bandwidth between the L2 cache and main memory is shared by all the processing cores [16, 50]. Multi-core examples are the IBM Power8 architecture [5], the Intel Xeon processors [7], and the AMD Opteron/AMD FX family [4].

The parallel programming model is used to develop parallel applications on the multi-core system. This model is an abstraction of the multi-core computer system architecture [29]. The following describes the main parallel programming models using a small set of features: implementation of the programming model, workload partitioning, synchronisation, and parallel programming constructs or patterns. The first three features were selected to constitute a full list of criteria [43], and the fourth feature was inspired by [50]. The programming model implementation is concerned with how parallelism capabilities are made available to programmers, for example, through thread libraries, template libraries, compiler directives, and language extensions. Workload partitioning is concerned with how data or workloads are partitioned into small chunks. Workload partitioning can be done either explicitly (data are partitioned manually by programmers) or implicitly (the programmers specify only which parts of the code can be processed in parallel, and the partitioning is realised automatically by the programming model). Synchronisation is concerned with the time sequence in which threads or tasks access shared data. Synchronisation can be done either explicitly (the programmers are responsible for writing code for thread management to provide conflict-free access to shared data) or implicitly (no programming effort is required from the programmers, and synchronisation constructs are available).

The fourth feature of parallel programming constructs or patterns involves those patterns that are valuable building blocks and how they are used in the design and implementation of parallel algorithms. Common and representative parallel patterns used in parallel programming are the fork-join pattern, the map pattern, the reduction pattern, the scan pattern, and the pipeline pattern [50]. In the fork-join pattern, the master thread creates several multiple threads that execute concurrently until they synchronise with the master thread at the end of execution. The map pattern applies a function or operation concurrently to every element of a data structure or a set of data structures with the same shape and usually creates a new data structure (or set of data structures) with the same shape as the input. The reduction

pattern applies an associative operation to all the elements of a data structure, reducing it to a single element. The scan pattern calculates all partial reductions of a data structure. The pipeline pattern consists of a linear sequence of tasks or threads and works as a producer-consumer relationship. Data flow through the pipeline from the first task to the last task. Each task implements one stage of the pipeline and performs a transformation on the output data of its predecessor. All these patterns are supported directly in one or more of the programming models discussed in this paper, or they can be implemented using other features if the patterns are not supported directly. The "from the inside" implementation details of these patterns are discussed in [50].

3.1. Multi-core programming frameworks

In this paper, the representative multi-core programming frameworks described below were included for three reasons. First, these frameworks were selected to cover a variety of options for most features of programming models, as discussed above. Second, these programming models were selected to cover a range of levels of programming abstraction for parallelism. This range of levels of abstraction helps to evaluate the programming effort required by each programming framework in a fair way. Finally, only frameworks that are used frequently by many programmers, especially for small multi-core platforms rather than many-core platforms, were considered. This approach ensured that quantitative and qualitative comparison of the multi-core programming frameworks was homogeneous.

Pthreads [25] is a common portable API and low-level thread library. The Pthreads library provides explicit low-level functions for creating and destroying threads and for coordinating threads using several synchronisation mechanisms. Finally, this model directly supports the fork-join programming construct.

OpenMP [3] is a very popular and portable API which uses implicit compiler directives to define parallelism or implicit data partitioning of *for* loops, work sharing, and synchronisation. OpenMP also provides some library functions for accessing the runtime environment. OpenMP supports the map and reduction patterns directly using compiler directives.

The Intel Cilk Plus [1] language is based on technology from Cilk [21], a parallel programming model for the C language. The Cilk Plus language provides keywords for implicit data partitioning or loop parallelism and implicit synchronisation. Finally, Cilk Plus supports the fork-join, map, and reduction patterns directly using keywords.

Intel TBB [2] is an open-source template library that offers a rich methodology for expressing regular and irregular parallelism in C++ programs. This template library consists of data structures or containers and algorithms that abstract the use of native threading packages in which individual execution threads are created, synchronised, and terminated manually. TBB emphasises an implicit data parallel programming model, enabling multiple threads to work on different parts of a data collection and enabling scalability to many cores. Moreover, TBB provides direct template functions for a variety of parallel patterns, including fork-join, map, reduction, scan, and pipeline.

SWARM [16] is an open-source parallel programming library. This library provides basic primitives for parallelisation, restricting control of threads, allocation and de-allocation of shared memory, and communication primitives for synchronisation, replication, and broadcasting. SWARM also supports the map, reduction, and scan patterns using implicit primitives.

FastFlow [10, 11] is an open-source, C++ template-based parallel programming framework for developing efficient applications for multi-core computers. FastFlow is conceptually designed as a stack of layers that progressively abstract the shared memory parallelism at the core level up to the definition programming constructs supported by structured parallel programming on shared-memory multi-core platforms. FastFlow provides 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. On the other hand, the map pattern is not supported directly in FastFlow, but can be implemented using the task-farm parallel pattern, as will be discussed later.

Table 2 summarises the features of the multi-core programming models discussed in this paper.

4. Parallel numerical and scientific computations

This section presents a set of basic scientific computations from numerical linear algebra (matrix computations and solving linear systems) and discusses general guidelines for parallelising these operations using parallel patterns irrespective of the parallel programming model used. Finally, several parallel programming models (as mentioned in Section 3) for parallelising scientific computations are also discussed.

Feature - Model	Pthreads	OpenMP	Cilk Plus	TBB	SWARM	FastFlow
Implementation	Library	Compiler	Language	Library	Library	Library
Data Partitioning	Explicit	Implicit	Implicit	Implicit	Implicit	Explicit
Synchronization	Explicit	Implicit	Implicit	Implicit	Implicit	Explicit
Parallel patterns	fork-join	fork-join map reduction	fork-join map reduction	fork-join map reduction scan pipe	map reduction scan	farm pipe

Table 2: Features of multi-core programming models

4.1. Matrix computations and related methods

Table 3 presents the matrix computations and related methods used to evaluate the multi-core programming frameworks. Note that the size of the vectors is n elements, whereas the size of the matrices is $n \times n$ elements. In addition, computation is measured by the number of floating-point operations, whereas memory is measured by the number of memory references (load/store operations). Furthermore, standard algorithms are used to measure computation operations and memory references for matrix computations, as shown in Table 3.

Matrix computations			
Computational kernel	**Operation**	**Computation**	**Memory**
Vector addition	$c_i = a_i + b_i, 1 \leq i \leq n$	n	$3n$
Inner or dot product	$z = x^T \cdot y = \sum_{i=1}^{n} x_i \cdot y_i$	$2n$	$2n + 1$
Matrix addition	$c_{ij} = a_{ij} + b_{ij}, 1 \leq i, j \leq n$	n^2	$3n^2$
Outer product	$z_{ij} = x \cdot y^T = x_i \cdot y_j, 1 \leq i, j \leq n$	n^2	$n^2 + 2n$
Matrix - vector product	$y_i = \sum_{j=1}^{n} a_{ij} x_j, 1 \leq i \leq n$	$2n^2$	$n^2 + 3n$
Matrix transpose	$a^T_{ji} = a_{ij}, 1 \leq i, j \leq n$	n^2	$2n^2$
Matrix product	$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}, 1 \leq i, j \leq n$	$2n^3$	$3n^2$
Solving linear systems			
Computational kernel	**Operation**	**Computation**	**Memory**
Gaussian elimination	division: $l_{ik} = a^k_{ik} / a^k_{kk}, k + 1 \leq i \leq n$ elimination: $a^{k+1}_{ij} = a^k_{ij} - l_{ik} a^k_{kj}$ $b^{k+1}_i = b^k_i - l_{ik} b^k_k, k \leq i, j \leq n$	$\frac{2}{3}n^3$	$n^3 + n^2$
Jacobi method	$x^{(k)}_i = \frac{1}{a_{i,i}}(b_i - \Sigma_{j \neq i} a_{i,j} x^{(k-1)}_j), 1 \leq i \leq n$	$4n^2$	$k(n^2 + 3n)$

Table 3: Some matrix computations and related methods

To achieve a fair comparison of multi-core programming models for implementing linear algebra computations, the matrix operations and related methods listed in the table were chosen based on the following criteria. First, these computations are used by most computational scientists as representative kernels in many important application domains such as computational statistics [34], combinatorial optimisation [41, 68, 70], and machine learning [26, 39]. Second, these computations were included to cover a range of ratios of computations to memory references, from low to high. Third, these computations cover different programming levels from easy to difficult from the coding perspective to evaluate model programmability. Finally, given that six programming models were studied, it was important to limit the amount of time spent on each method and operation.

Except from the standard algorithms, there are alternative methods for calculating a matrix product, which require fewer arithmetic operations by using the divide and conquer technique. Strassen's method and one of its variants, that of Winograd and Coppersmith, were the first of these algorithms and have an asymptotic complexity of $O(n^{2.807})$ and $O(n^{2.376})$ respectively [28, 35]. Furthermore, interesting research has been performed on cache-oblivious algorithms, which analyse cache complexity for linear algebra operations like matrix transpose, matrix product, and Gaussian

elimination [32, 33]. Finally, there are matrix-vector product algorithms, which take $O(n \log n)$ time using the fast Fourier transform (FFT) family of algorithms [35].

Matrix operations such as matrix transpose, matrix product, and Gaussian elimination can be implemented effectively on a computer with caches using tiling (or blocking) techniques [35] to increase memory access efficiency. These tiling techniques have been implemented for some linear algebra operations in the popular LAPACK [14] and PLASMA software packages [9]. Another optimised BLAS library, GotoBLAS [37], elaborately identifies tiles in a matrix. GotoBLAS optimises matrix multiplication and increases memory access efficiency further by copying the most frequently used data into contiguous memory locations to reduce translation look-aside buffer (TLB) table misses, an issue that results in significant performance degradation, but is generally not addressed by other BLAS routines. Finally, there are Intel Math Kernel Library (MKL) versions of BLAS and LAPACK [6], which use the Intel architecture's SSE instruction extensions heavily to perform matrix computations in the SIMD mode.

Likewise, there are two categories of methods for solving linear systems: direct methods (i.e., Gaussian elimination and the Gauss-Jordan method) [60] and iterative methods (i.e., Jacobi, Gauss-Seidel, and successive over-relaxation) [17]. The choice of solution method for a linear system depends on its structure and size. Roughly speaking, direct methods are best for small and full (dense) matrices, whereas iterative methods are best for very large and sparse matrices. Furthermore, some important pivoting strategies are used in Gaussian elimination, such as full pivoting, column pivoting, row pivoting, and no pivoting [60]. For some classes of matrices, such as diagonally dominant matrices, no pivoting is required. For this research, the commonly used method for each category was selected, i.e., the direct Gaussian elimination method with no pivoting and the Jacobi iterative method.

This paper focusses on standard algorithms for matrix computations and related methods, as shown in Table 3, for the following reasons. First, standard algorithms were selected to keep the same algorithmic and implementation style for all multi-core programming models by the greatest number of programmers. In this way, a fair qualitative comparison of the programming models implementing advanced algorithmic techniques was ensured. Implementation of advanced algorithmic techniques can lead to different codes by programmers with different levels of experience, and qualitative comparison of these programming models may not produce meaningful conclusions. Second, these algorithms for matrix computations and related methods were implemented so that the parallelisation approaches would be comparable and have the same structure, simplicity, and portability for all multi-core programming models. Finally, the present study focusses on ordinary algorithms and simple parallelisation techniques from the perspective of a moderately experienced programmer. These reasons were taken into account so that the multi-core programming models could be compared in a homogeneous way.

4.2. Parallelisation using parallel patterns

A brief description of the parallelisation of matrix computations and related methods in terms of parallel patterns is given below.

Consider the vector addition $c = a+b$. Each output element of vector c in a vector addition operation is independent of all other elements, and therefore the serial loop can be parallelised using the map parallel pattern. In this case, the map pattern applies the addition operation to each element of vectors a and b. This operation applies as many times as there are elements in vectors a and b.

Consider the dot product $z = x^T \cdot y$. The dot product operation has a loop-carried dependency, and therefore it cannot be parallelised. For this reason, the serial kernel operation is reorganised so that each loop iteration calculates an initial pairwise product, and at the end of the loop, these products are summed. This modified kernel can be parallelised using the map parallel pattern combined with a reduction pattern. In this case, the map pattern applies the initial product to each element of vectors x and y, and the reduction pattern applies the summation of the corresponding products.

Consider the matrix addition $C = A + B$. Each element of the output matrix C computed in a matrix addition operation is independent of all other elements, and the serial double loop can be parallelised at the coarse-grained level by applying the map pattern to the outer loop. In this case, the map pattern applies the addition operation to each row of matrices A and B. This procedure applies as many times as there are rows in matrix A.

Consider the outer product $Z = x \cdot y^T$. Each element of the output matrix Z computed for an outer product operation is independent of all other elements, and therefore the serial double loop can be parallelised at the coarse-grained level by applying the map pattern to the outer loop. In this case, the map pattern applies the product operation between an element of vector x and the entire vector y. This procedure applies as many times as there are elements in vector x.

Consider the matrix-vector product $y = Ax$. Each output element of vector y for the matrix-vector product is considered as the computation of the dot product between a row of matrix A and vector x. All output elements of vector y are calculated independently, and therefore the serial double loop can be parallelised by applying the map pattern to the outer loop. In this case, the map pattern applies the dot product operation between a row of matrix A and vector x. This procedure applies as many times as there are rows in matrix A.

Consider the matrix transpose $A^T = A$. Each column of the transposed matrix A^T in a matrix transpose operation is considered to be a redistribution of the elements of a row in matrix A. All columns of matrix A^T are calculated independently, and therefore this serial double loop can be parallelised by applying the map pattern to the outer loop. In this case, the map pattern redistributes the elements of a row of matrix A. This procedure applies as many times as there are rows in matrix A.

Consider the matrix product $C = AB$. Each row of the output matrix C of a matrix product operation can be calculated as the product between a row of matrix A and the whole matrix B. Therefore, the serial triple loop can be parallelised at the coarse-grained level by applying the map pattern to the first outer loop. This procedure applies as many times as there are rows in matrix A.

The Gaussian elimination operation is a direct method for solving dense linear systems of equations such as $Ax = b$, where A is a known non-singular $n \times n$ matrix with non-zero diagonal entries, b is the right-hand side, and x is the vector of unknowns. This discussion will focus only on the computational part of Gaussian elimination, which transforms the matrix A into triangular form with an accompanying update of the right-hand side, so that the solution of the system is straightforward by a triangular solution method. Each step k of Gaussian elimination requires a division operation followed by $n - k$ elimination operations, as shown in Table 3. These division and elimination operations can be parallelised by applying the map pattern to the corresponding *for* loops. In this case, the map pattern of the division operation applies the division operation to each element of a column of matrix A, whereas the map pattern of the elimination operation applies the elimination operations to each row of matrix A and the corresponding element of vector b. This procedure applies as many times as there are rows in matrix A. Note that at the end of the division operation and the elimination phase of step k, there is a synchronisation barrier. This synchronisation is necessary so that the operations of the current elimination step are completed before the start of the next elimination step [19].

The Jacobi method is an iterative method for solving linear systems and can be expressed as an iterative scheme consisting of a serial matrix-vector product followed by a solution vector update. Therefore, parallelisation of the Jacobi method corresponds to parallelisation of the matrix-vector product using the map pattern, as discussed earlier.

4.3. Parallel implementation issues

When parallelising the linear algebra operations described above on a multi-core system, a simple block partitioning technique was used to increase the amount of work per thread. More specifically, scientific computations that use vectors partition a vector into blocks of equal size, i.e., $\lceil n/p \rceil$ (where n is the number of elements in the vector and p is the number of cores), so that each core operates on a block of $\lceil n/p \rceil$ elements. On the other hand, linear algebra operations that use matrices partition a matrix into blocks of rows of equal size, i.e., $\lceil n/p \rceil$, so that each core operates on a block of rows. In other words, parallelisation of linear algebra operations that use vectors is based on parallelisation of a *for* loop, whereas parallelisation of linear algebra operations that use matrices is based on parallelisation only of the outer *for* loop. Details of how the map and reduction patterns and the synchronisation barrier are expressed in several parallel programming models for parallelising linear algebra operations are presented below. Note that only moderate programming expertise is required to implement the linear algebra methods because only simple code is written and the built-in high-level programming constructs provided by each model can be used.

There are no built-in constructs for expressing map and reduction patterns such as the synchronisation barrier in the Pthreads model. For this reason, corresponding codes for implementing these map and reduction patterns and the barrier had to be written. To implement a map pattern with Pthreads, a number of threads are launched by calling the `pthread_create` function, which passes two parameters: a function pointer and a pointer to that function's arguments. The call to function `pthread_create` creates a thread that runs the function and passes the specified arguments to it (such as thread rank and data structures) [44]. For parallel implementation of the corresponding linear algebra operation, a corresponding function was created that embraces the original serial body code of the corresponding linear operation, but the bounds of the loop were modified so that each thread works on its own local data block.

The bounds of the loop are based on thread rank as follows: each thread works on its own block of data $b = \lceil n/p \rceil$, which ranges from $rank \times b$ to $(rank + 1) \times b$. Finally, the threads are synchronised at the end of the calculation by calling the `pthread_join` function. To implement a reduction pattern with Pthreads, the low-level explicit synchronisation function `pthread_mutex_lock` was used. To express the barrier in the Pthreads model, a linear barrier was implemented as an individual function using the functions `pthread_mutex_lock, pthread_mutex_unlock, pthread_cond_broadcast`, and `pthread_cond_wait`.

The map and reduction patterns can be expressed in the OpenMP model using the built-in constructs `#pragma omp parallel for` and `#pragma omp reduction`. The code following the `#pragma omp parallel` for a compiler directive (i.e., the parallel region) is run by a number of parallel threads, and loop iterations are distributed to threads executing in the parallel region. At the end of the loop, there is an implicit barrier. Moreover, distribution of loop iterations to threads is done by an OpenMP scheduling strategy. OpenMP supports various scheduling strategies specified by the `schedule` clause. In this case, no strategy was specified in the schedule clause, and therefore the OpenMP model created evenly sized blocks of loop iterations for all threads in the parallel region. The `#pragma omp reduction` clause implements an automatic parallel reduction of the variables that appear in its list with a predefined associative operator. To implement the sum reduction in the dot product operation, a *combine* operator (addition) was specified. Finally, the built-in `#pragma omp barrier` was not used to define a synchronisation barrier because there is an implicit barrier at the end of the parallel *for* loop.

Cilk's `cilk_for` and `reducer_opadd` keywords perform map and reduction patterns respectively. The `cilk_for` keyword processes loops in parallel and divides each loop into equal-sized blocks containing one or more loop iterations. Each block is executed serially and is spawned as a block during loop execution. The maximum number of iterations in each block is the grain size. The `#pragma cilk_grainsize` clause was also introduced at the beginning of the loop, and the grain size for one loop was specified as $\lceil n/p \rceil$ according to the proposed data partitioning strategy. Moreover, implementation of the `cilk_for` construct is based on a recursive fork-join model, which spreads the overhead of managing the map over multiple threads. Finally, the reduction pattern can be expressed in Cilk Plus by a hyperobject. This hyperobject is a reducer, which eliminates 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. Cilk's reducers work for any operation that can be re-associated. To implement the sum reduction, the hyperobject `reducer_opadd` was used. Finally, the synchronisation barrier in this model was implemented by an implicit barrier at the end of `cilk_for`.

The map and reduction patterns can be implemented in TBB using the built-in template functions `parallel_for` and `parallel_reduce` respectively. To parallelise linear algebra computations with the TBB model, the original outer loop was replaced 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 lambda function. This function partitions the iteration space into subspaces and applies the lambda function to each subspace in parallel. The iteration space is defined by three parameters: the lower bound, the upper bound, and the optional increment. The lambda function is a new feature introduced by the C++11 standard and is widely supported in C++ compilers. The lambda function is an alternative way to create the equivalent function object, which would otherwise be written by hand. The lambda function has three parts: the capture part, a parameter list, and the function definition. The capture part describes how to capture local variables outside the lambda, i.e., by value or by reference. The argument list takes an argument of type `blocked_range<T>`, which defines an iteration range. The function definition takes the original loop or code of the corresponding linear algebra method; the loop's iteration bounds are specified dynamically by the argument passed to the lambda function [50, 44, 62]. Finally, the reduction pattern can be expressed using the template function `parallel_reduce`, which performs a parallel reduction over a recursive range. This template function has two forms: the functional form is designed to be easy to use in conjunction with lambda expressions, and the imperative form is designed to minimise data copying. Here, the functional form of *reduce* was used, which requires four arguments: the iteration space as in `parallel_for`, the identity element of type `T`, e.g., `double`, a functor provided by the user for the serial reduction, and finally a binary operator reduction to combine the results of the user functor, e.g., `plus<double>` for summation of double elements [50].

The map and reduction patterns can be expressed in SWARM using the built-in constructs `SWARM_pardo` and `SWARM_reduce` respectively. To implement linear algebra methods in the SWARM framework, the original outer *for* loop was replaced 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 synchronisation of communication between cores. Moreover, the sum reduction was replaced by a function `SWARM_reduce`, which is used to add values calculated by different threads. This function requires three arguments: the local variable of each thread, the predefined reduction operator (e.g., `SUM`) that is performed on these variables, and a built-in structure `THREADED` that contains all the required information for the particular thread. Finally, the function `SWARM_Barrier_sync` was used to express the synchronisation barrier.

The FastFlow model does not provide ready-to-use constructs for the map and reduction patterns, and therefore code had to be written to implement these patterns explicitly using other features such as an accelerator with the farm template and spinlocks. To parallelise linear algebra methods for the map pattern, the FastFlow accelerator was used, which is a software device that can be used to speed up skeleton structured portions of code using the cores left unused by the main application. In other words, FastFlow accelerates particular computations by using a skeleton program, offloading to it the tasks to be computed. For this reason, a skeleton program was written using the FastFlow skeletons (i.e., the farm template), which computes the tasks that will be given to the accelerator. This skeleton program, when used to program the accelerator, must have an input stream, which is used to offload tasks to the accelerator. Then the skeleton program must be run using a method such as `run_then_freeze()`. This method will start the accelerator skeleton program, consuming the input stream items either to produce output stream items or to consolidate (partial) results in memory [10, 11]. Finally, a new class task was created for each linear algebra method, with members corresponding to the variables that are outside the scope of the loop to be parallelised. The class must overload the function task method, and it takes an argument such as task rank. The method's body takes the original outer loop or code of the corresponding linear algebra method; the loop's iteration bounds are based on task rank by the argument passed to the function task method, so that each task works on its own block of data. Moreover, to implement the reduction pattern with FastFlow, the low-level explicit synchronisation function `spin_lock` was used. Finally, the FastFlow method `wait()` was used to express the synchronisation barrier.

5. Quantitative and qualitative results

This section presents the results of an extensive quantitative and qualitative evaluation performed to gain insight into the practical behaviour of each of the reviewed programming models used to implement the scientific computations.

5.1. Quantitative comparison

For quantitative or performance comparison, computational experiments were performed. The experiments were run on a 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 this research project. The implementations of the serial and multi-thread algorithms for the nine linear algebra operations were developed in the ANSI C/C++ programming language for all the reviewed multi-core programming models. To compile the multi-thread programs, the Intel C/C++ compiler version 13.0, i.e., `icc, icpc`, was used because it is a very widely used compiler.

The quantitative evaluation was organised into two parts:

1. Examination of the parallel performance (in terms of execution time) of the linear algebra methods for all the reviewed multi-core programming models. More specifically, performance was examined in two ways: (a) as a fixed-size problem with increasing number of cores, and (b) with a fixed number of cores where the matrices or problems increase in size. Moreover, the optimal performance of each computational kernel was examined against the programming model. These performance tests used parallelisations of the optimised sequential implementations.
2. Examination of the overall performance of all the reviewed programming models as a function of number of cores and of matrix size.

A set of randomly generated input matrices and vectors with sizes ranging from 1024×1024 to 7168×7168 in double-precision format was used to evaluate the performance of the multi-thread linear algebra methods. The elements of the generated input matrices for the matrix computations were chosen from a uniform distribution on the interval $(0.0, 100.0)$, whereas the input matrices for the methods for solving linear systems were created to be strictly

row-diagonal dominant. For experimental comparison of the results, the dense and sparse matrices can be used as test cases for all matrix computations and related methods, but this study focusses on dense matrices, which serves the goals of this paper.

To assess the performance of the multi-thread linear algebra methods for all programming models, practical execution time was used as a measure. The practical execution time is the total time that a multi-thread algorithm takes to complete a computation. The execution time is obtained by calling the C function `gettimeofday()` and is measured in seconds. To decrease random variation, the execution time was measured as an average of ten runs. A models performance for each computational kernel was calculated using the relation

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

where $BT(model)t$ is the best execution time for a specific computational kernel for all six models and $T(model)i$ is the execution time of model i for the same computational kernel. The best time was then set equal to 100%. To calculate the overall performance for each model for each combination of matrix size and number of cores, the percentage values for all computational kernels were added up, and the sum was divided by the total number of computational kernels, i.e., nine. Finally, the highest percentage corresponded to the best overall performance.

The figures following this section use data extracted from the parallel performance experiments. These figures are performance graphs for all computational kernels using all programming models. Note that the performance results are based on parallelisation of the optimised sequential implementations with flag O2, which includes vectorisation. Figure 1 (or Tables 10-18 in the appendix) presents the execution times for all computational kernels as a function of the number of cores for a fixed matrix size of 7168, whereas Figure 2 (or Tables 28-36 in the appendix) presents the execution times for all computational kernels as a function of problem size for a fixed number of cores (16). Note that time on the y-axis in Figures 1 and 2 has a logarithmic scale. Figure 1 includes a line for the serial execution time, which is the time taken by a pure C sequential program. Finally, the tables also report the bootstrap confidence intervals of the mean execution time for all computational kernels.

From the graphs in Figure 1, it is clear that there was little overhead for the programming models on a single core, except for the vector addition and dot product operations. The vector addition and dot product operations had the greatest overheads for all programming models and all numbers of cores because of the limited amount of parallelisable work available. Pthreads, Cilk Plus, TBB, and FastFlow incurred some overhead for all linear algebra methods (except for vector addition and dot product) on a single core, but all six multi-core implementations showed identical performance with few exceptions when multiple cores were used. This overhead was due to use of the `pthread_create` and `pthread_join` functions in the Pthreads model to create and terminate threads, use of automatic partitioning of the `parallel_for` template function in the TBB model, which incurs an overhead cost for every chunk of work that it schedules, use of the `cilk_for` construct in the Cilk Plus model, which is based on the recursive fork-join model, and the implicit barrier at the end of the loop and the use of additional internal synchronisation mechanisms in the FastFlow model to manage tasks. Note that a partitioner was not declared in the third optional argument of the `parallel_for` function of the TBB model, and therefore TBB used automatic partitioning. The OpenMP and SWARM implementations of the other seven linear algebra methods (except for vector addition and dot product) had very little overhead for one core and scaled best for all numbers of cores. Note that for the matrix product operation, no overhead cost was incurred for any of the programming models.

Furthermore, from the graphs in Figure 1, the execution time of all programming models for all computational kernels decreased as the number of cores increased, with the exception of the vector addition and dot product operations. More specifically, the execution times of the matrix addition, outer product, matrix-vector product, matrix product, Gaussian elimination, and Jacobi operations decreased significantly compared to the execution time of the matrix transpose operation. The limited rate of decrease in the execution time of the matrix transpose operation occurred because this operation has poor spatial locality and scans the matrix column by column instead of row by row, leading to a higher cache miss rate. However, there was a saturation point with eight cores for all programming models for the matrix addition, matrix-vector product, Gaussian elimination, and Jacobi operations, after which the execution time was stable. This occurred because the amount of parallelised work for the matrix addition and matrix-vector operations was small with a large number of cores because these operations are medium-scale. Furthermore, the constant performance of the Gaussian elimination and Jacobi methods with a large number of cores (i.e., more than eight) was observed because Gaussian elimination requires more synchronisation with a large number of cores and because

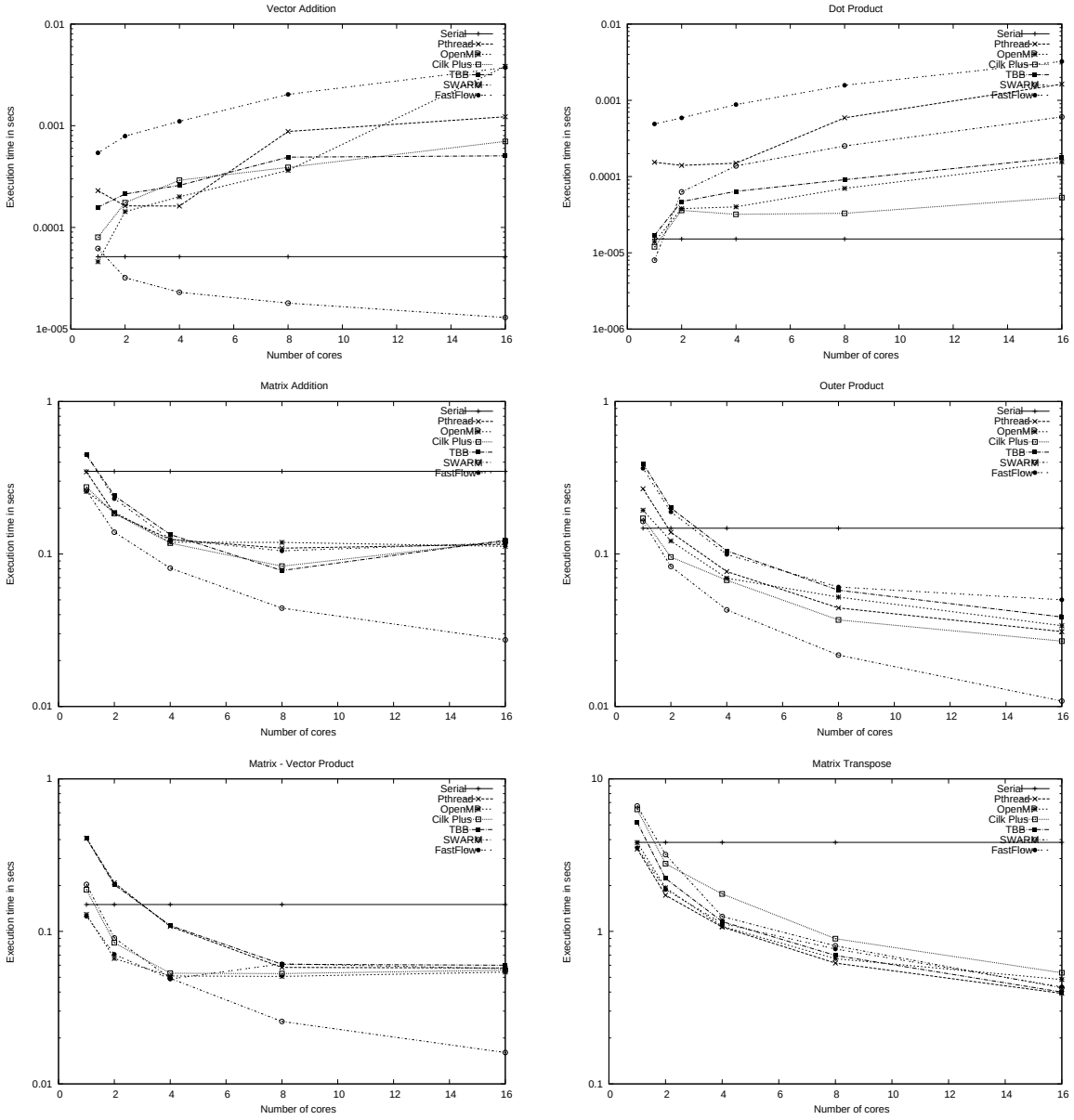

Figure 1: Execution times (in secs) of the scientific computing kernels as a function of the numbers of cores for a fixed problem size of 7168

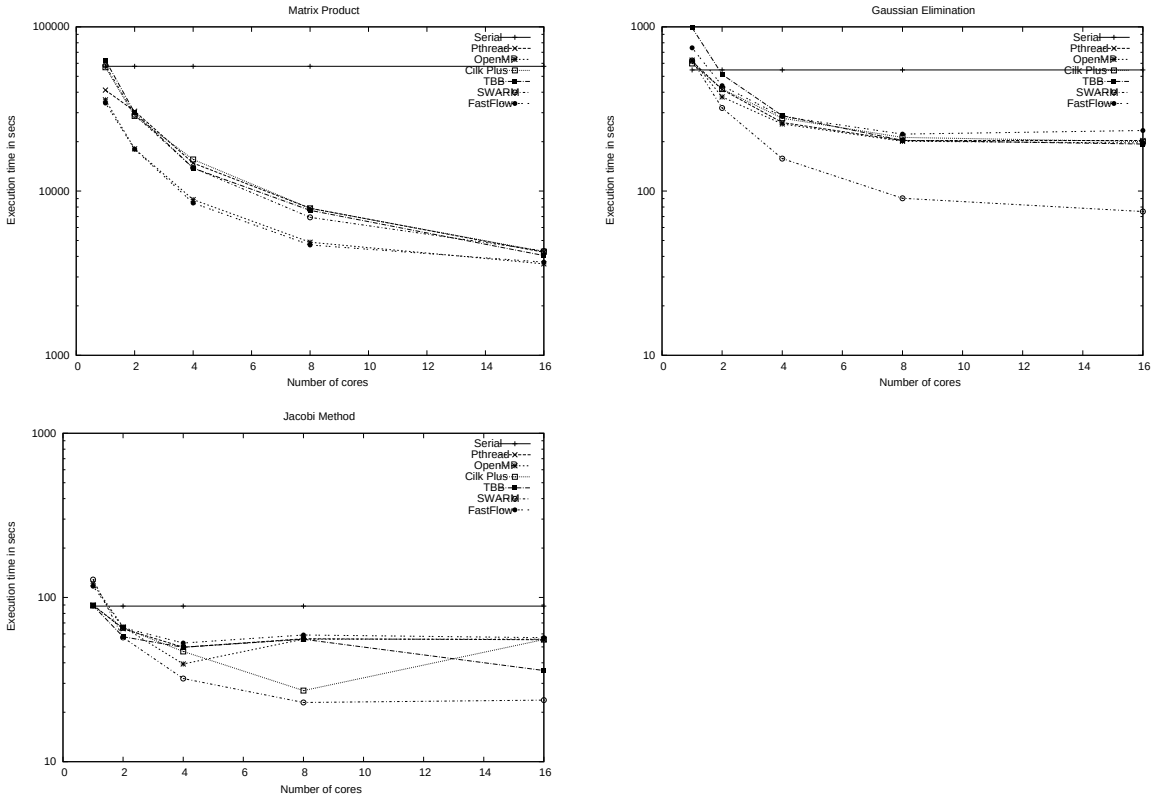

Figure 1: (continued) Execution times (in secs) of the scientific computing kernels as a function of the numbers of cores for a fixed problem size of 7168

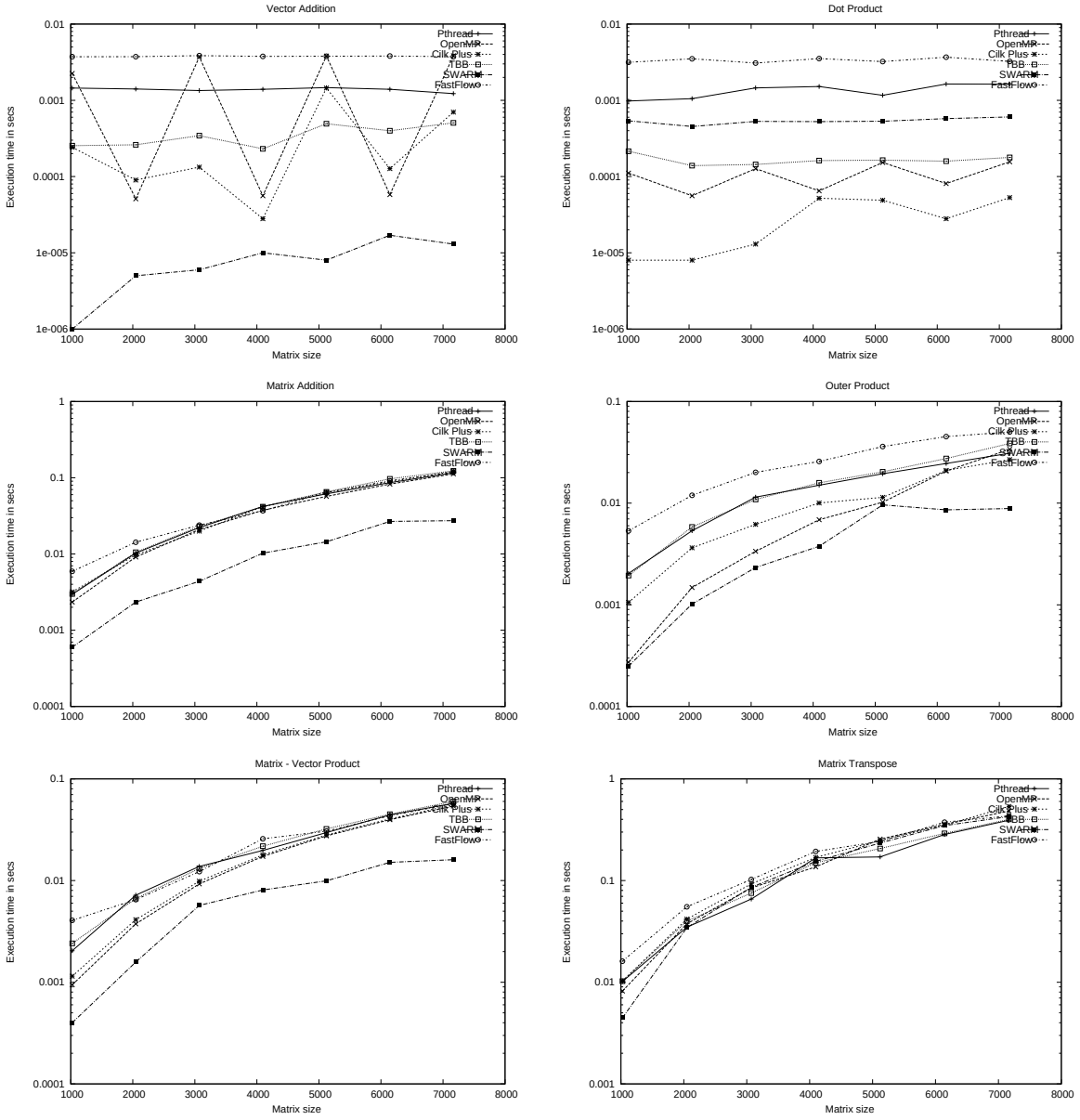

Figure 2: Execution times (in secs) of the scientific computing kernels as a function of the problem size for 16 cores

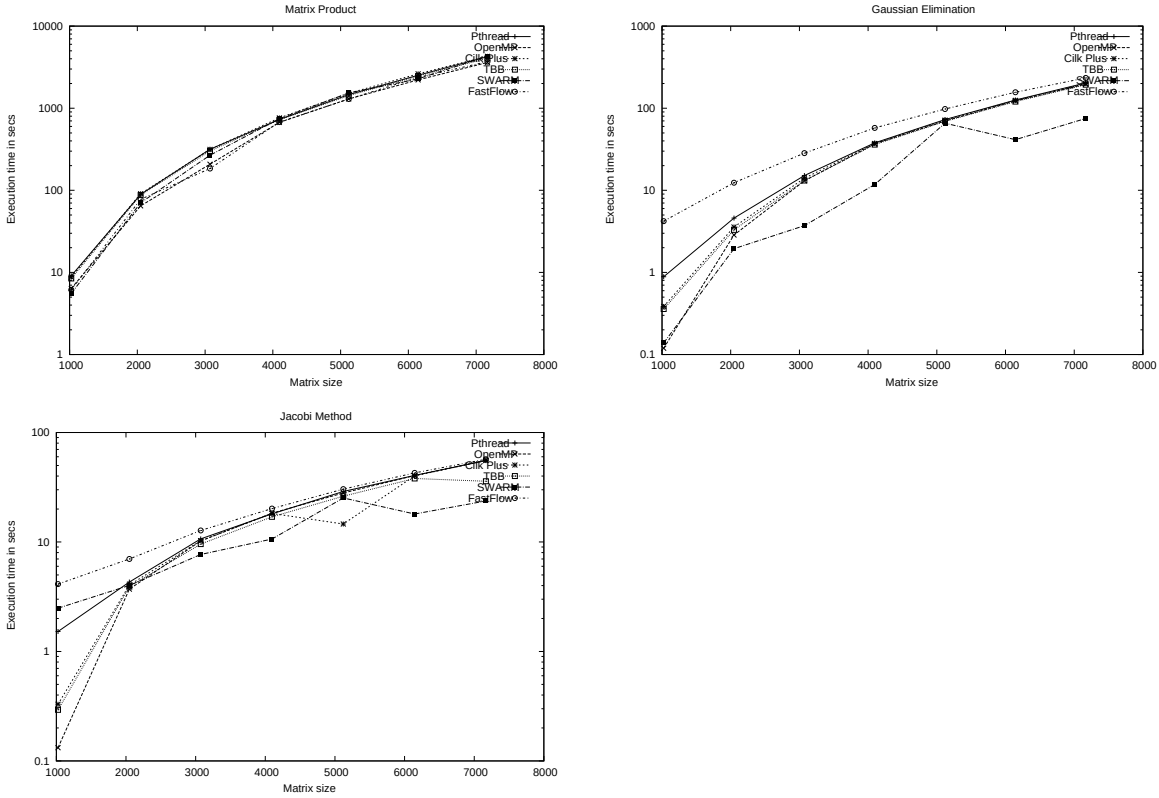

Figure 2: (continued) Execution times (in secs) of the scientific computing kernels as a function of the problem size for 16 cores

the Jacobi method uses the matrix-vector operation as a medium-scale operation, as explained earlier. On the other hand, the execution time of the vector addition and dot product operations increased slightly as the number of cores increased. This happened because these operations require a small number of steps (n steps approximately) compared to the other six matrix operations, which require n^2 or n^3 steps.

Based on the graphs in Figure 1, some specific remarks can be made about performance. For the vector addition operation, the SWARM implementation had the best performance in terms of execution time for any number of cores, whereas the FastFlow implementation had the slowest performance. The poor performance of FastFlow was due to its use of a built-in software accelerator (i.e., a collection of threads) to implement the vector addition method, which provided additional synchronisation mechanisms (i.e., locks and waits) to manage tasks internally. This overhead for internal synchronisation mechanisms was clearly evident for a large number of cores compared to the small amount of computation needed for the vector addition method. This overhead can be eliminated by an experienced programmer because the FastFlow programming model has a moderately steep learning curve, as will be discussed later. Moreover, the performance of the remaining models was closest to that of the Pthreads implementation.

For the dot product operation, the performance of all parallel implementations was worse than the serial performance. The Cilk Plus model had the best performance among the poorly performing parallel implementations. The poorer performance of these models can be explained by their high overhead costs compared to the limited amount of parallelised work performed.

For the matrix addition operation, the OpenMP and SWARM implementations had the best performance in terms of execution time for small and large numbers of cores respectively. On the other hand, the performance of the other programming models was similar for a large number of cores, with the exception of the SWARM model. However, the Pthread, TBB, and FastFlow models had the poorest performance for any number of cores due to the overhead of creating and terminating threads in the Pthread model, the automatic partitioning in the TBB model, and the additional internal synchronisation mechanisms discussed earlier.

For the outer product operation, SWARM had the best performance in terms of execution time for any number of cores due to its low overhead. On the other hand, TBB and FastFlow had the worst performance for small and large numbers of cores respectively. The poor performance of the TBB model was due to additional implicit partitioning overhead, and the poor performance of FastFlow was due to synchronisation overhead, as discussed earlier. Note that the amount of synchronisation overhead was large with respect to the amount of parallelisable work available. For the remaining programming models, their performances were approximately the same for any number of cores.

For the matrix-vector product, the OpenMP and SWARM implementations had the best performance in terms of execution time for small and large numbers of cores respectively. The SWARM implementation performed best, with a large improvement compared to the other models, for a large number of cores because the system runtime overhead was spread over multiple cores. The performance of the remaining models was identical.

For the matrix transpose operation, Pthreads had satisfactory performance for any number of cores. The performance of the remaining implementations was closest to those of Pthreads and SWARM.

For the matrix product operation, the OpenMP and FastFlow implementations had satisfactory execution time for any number of cores. However, a low rate of decrease was observed in the execution time of the FastFlow model. From these two performance observations, the following could be concluded: the synchronisation overhead for FastFlow with a small number of cores was negligible compared to the high amount of parallelised work for the matrix product, whereas the synchronisation overhead was clearly evident for a large number of cores compared to the small amount of parallelised work performed. Finally, the performance levels of the remaining implementations were identical for any number of cores.

Finally, for the Gaussian elimination and Jacobi operations, the SWARM implementation had satisfactory performance for any number of cores. The performance values of the remaining implementations were identical for any number of cores.

The ANOVA statistical test was used to verify the statistical significance of these results. Tables 1-9 in the appendix show the ANOVA results for all matrix operations. As can be seen in the tables, the factors (programming model and number of cores) and their interaction (programming model $\times$ number of cores) were statistically significant (p-value < 0.05).

Based on the graphs in Figure 2, it is apparent that the execution time of all programming models for all computational kernels increased with increasing matrix size, with the exception of the vector addition and dot product operations. More specifically, the execution times of the matrix addition, outer product, matrix product, matrix trans-

pose, Gaussian elimination, and Jacobi operations increased significantly, whereas that of the matrix-vector product increased only slightly.

Based on the graphs in Figure 2, specific remarks about performance can be made. For the vector addition and dot product operations, the SWARM, OpenMP, and Cilk Plus implementations had good and constant performance for all problem sizes because they incurred low system overhead. On the other hand, Pthreads and FastFlow had poor performance because of the high amount of overhead compared to the amount of parallelisable work available, as discussed earlier.

For the matrix addition, outer product, and matrix-vector product operations, the SWARM implementation had the best performance in terms of execution time for any problem size. On the other hand, the Pthreads and FastFlow implementations had the poorest performance because of their system overhead. Note that the performance values of the Pthreads and FastFlow implementations of the matrix addition operation were closer to each other than to those of the other implementations. Moreover, the performance of the remaining implementations for the other two operations was modest, and the execution times of these implementations were very close.

For the matrix transpose operation, all implementations had good and identical performance for medium and large matrix sizes. The FastFlow implementation had the longest execution time, and the differences in execution times compared to the other implementations decreased as matrix size increased. This can be explained by noting that the synchronisation overhead for FastFlow becomes small when the matrix size is large.

For the matrix product operation, the performance of all programming models was excellent, and they scaled identically as a function of matrix size. For this kernel operation, the OpenMP and FastFlow implementations performed the best, with minimal differences from each other.

Finally, for the Gaussian elimination and Jacobi operations, the SWARM implementation had satisfactory performance for any problem size. The execution times of the remaining implementations were similar. The performance of Gaussian elimination for all models improved as matrix size increased because of the limited synchronisation (i.e., barrier) overhead when the matrix size was large.

The ANOVA results for the execution times for all matrix operations as a function of problem size for 16 cores are listed in Tables 19-27 in the appendix. As can be seen in the tables, the factors (programming model and problem size) and their interaction (programming model $\times$ problem size) were statistically significant (p-value < 0.05).

Figures 3 and 4 present the results from a preliminary scalability analysis for two selected matrix operations, the outer product and the matrix product. Figure 3 shows the speed-ups of these two matrix operations when the number of cores was increased for various problem sizes. Speed-up is defined as the serial reference runtime of the pure C sequential program when run on a single-core processor, divided by the time taken by the optimised multi-thread program on a multi-core processor. In this analysis, SWARM was used for the outer product and OpenMP for the matrix product because they performed well in the previous tests. These results show that a larger instance of the same problem yields higher speed-up and efficiency for the same number of cores, although efficiency continued to drop as the number of cores increased. Note that efficiency is a measure of the fraction of the time during which a core is usefully employed; it is defined as the ratio of speed-up to the number of cores [38].

Figure 4 shows the scalability of two matrix operations (the outer and matrix products) for all programming models. Here the problem size was increased at the same rate as the number of cores, so that the number of basic operations required by each core for each problem remained approximately constant. These results show that the efficiency of the two matrix operations was approximately constant for all programming models. Furthermore, the SWARM and OpenMP models achieved higher efficiency (from 0.80 to 0.99) than the remaining models for the outer product operation, whereas the other models except for FastFlow had efficiencies ranging from 0.55 to 0.70. However, the efficiency for the outer product operation fluctuated for some programming models. More specifically, the overhead percentages for the SWARM, OpenMP, Cilk Plus, Pthread, TBB and FastFlow models ranged from 1% to 9%, 3% to 24%, 5% to 27%, 1% to 15%, 7% to 25%, and 3% to 40% respectively as the number of cores increased. On the other hand, the efficiency for the matrix product was sufficiently high (from 0.80 to 0.90) for all programming models. In addition, the overhead percentage was sufficiently small (i.e., 3% to 30% for a large number of cores) compared to the efficiency of the outer product operation. Finally, an ANOVA analysis showed that these results were statistically significant.

Figure 5 presents the overall performance for all the reviewed programming tools as a function of number of cores and of matrix size. The average performance shown on the left of Figure 5 for each model and each number of cores is the median value for all computational kernels and matrix sizes. Similarly, the average performance shown on the

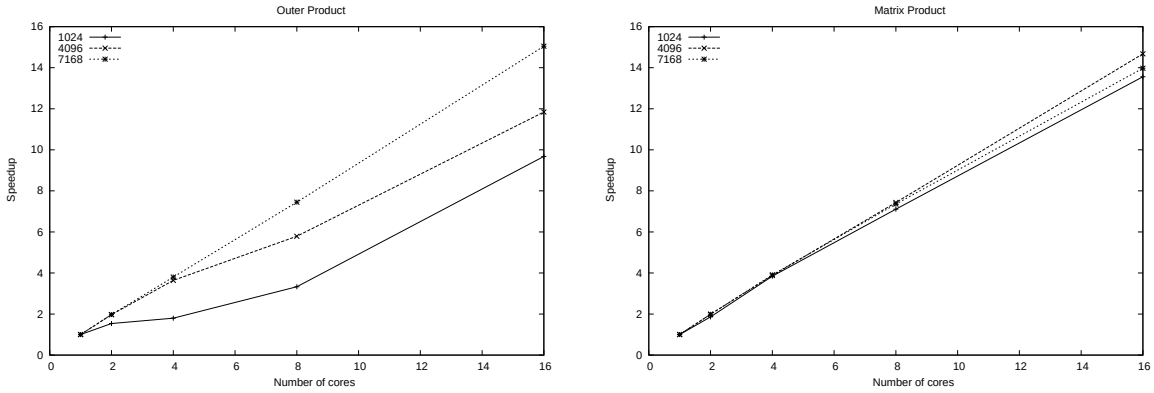

Figure 3: Speedups of two matrix operations versus the number of cores for different problem sizes using the SWARM model (left) and OpenMP model (right)

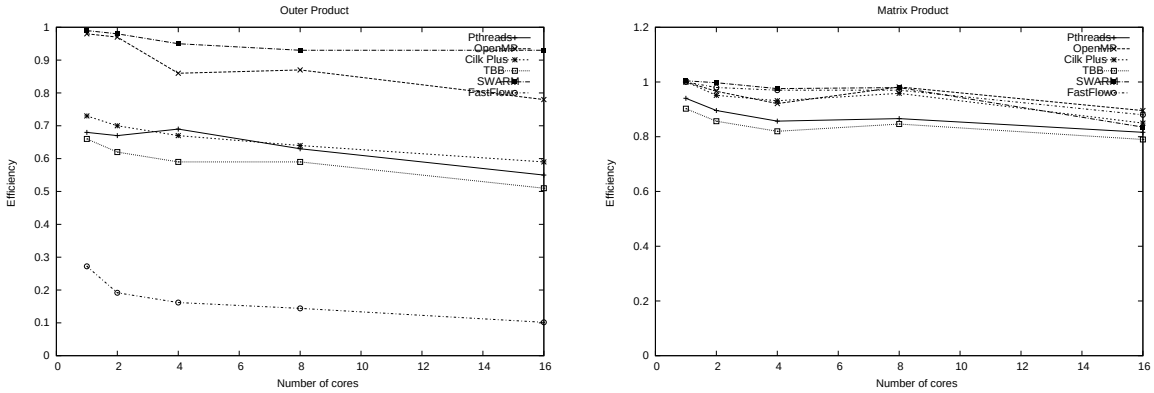

Figure 4: Efficiency of two matrix operations as a function of problem size and number of cores

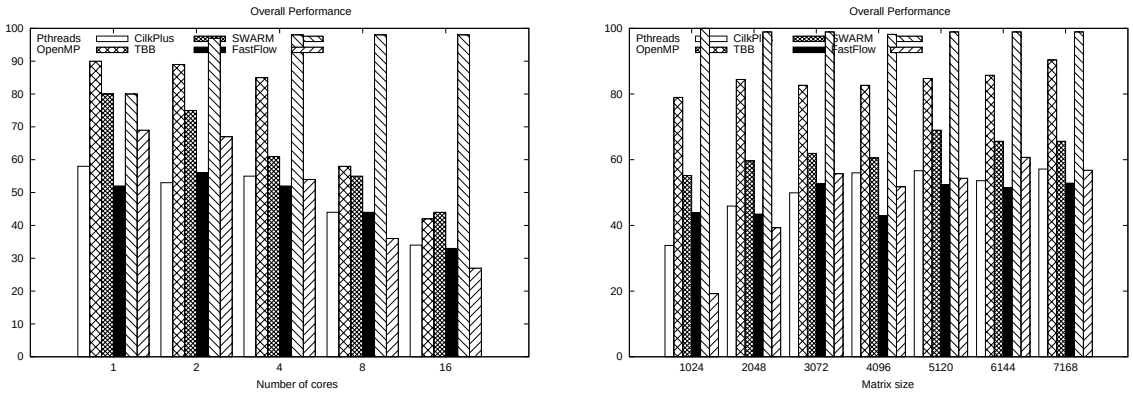

Figure 5: Overall performance as a function of the number of cores (left) and overall performance as a function of the problem size (right)

right of Figure 5 for each model and each matrix size is the median value for all computational kernels and numbers of cores.

Based on these results, OpenMP had the best performance with a small number of cores, whereas SWARM had the best performance for a larger number of cores and for any matrix size. However, the performance difference between the OpenMP and SWARM models became small as matrix size increased, whereas the difference remained large as the number of cores increased. Moreover, the overall performance of the SWARM and OpenMP models was outstanding, with large differences from the other models for a large number of cores, as presented on the left-hand side of Figure 5. The overall performance gap between SWARM and OpenMP and the other models was constant for all matrix sizes. In both cases, the SWARM and OpenMP models had the best performance because they appeared as a "winner" in most linear algebra operations (except for matrix product) and because they provide an optimised runtime system, unlike the other models. The next best performing models in order were Cilk Plus, TBB, and FastFlow, with very small differences for any number of cores and any matrix size. Finally, in last place came the Pthreads model, which had the poorest performance in most cases.

5.2. Qualitative comparison

Research has been carried out into the programming effort required for parallel programming [13, 42, 48, 61, 67]. Programming effort has been evaluated using basic metrics such as the time taken to write a program, the development time, or the number of lines of source code. However, it is not easy to determine the development time for coding a program, and the number of lines of code has been recognised as an inadequate measure [42, 64]. For this reason, these metrics (lines of code and development time) were combined here with a series of software engineering parameters such as library popularity, level of abstraction, programming style, support for online help and documentation, and learning curve to assess the intensity of programming effort required for the multi-core programming frameworks [30, 67]. Recent work [64] has identified more useful metrics for evaluating programming effort, but they require powerful and reliable tools for data gathering and analysis.

The number of additional or modified lines of code needed to parallelise the problem or function was counted in each case, excluding variable declarations. The development time included the time in days for parallelising and coding the problem in each multi-core programming framework by a user with moderate programming experience. Because it is difficult and time-consuming to measure development time, an empirical and qualitative rating scale was used, where the notation $+++$ means the shortest time, whereas the notation $---$ denotes the longest time [13]. Table 4 gives the number of additional or modified lines of code and the qualitative rating of development time for the nine computational kernels required by each framework.

Based on this table, the following remarks can be made: the OpenMP, SWARM, Cilk Plus, and TBB implementations required the same number of additional lines to parallelise a serial function. In the OpenMP and Cilk Plus

Computational kernel	Pthreads		OpenMP		Cilk Plus		TBB		SWARM		FastFlow	
	LOC	Time	LOC	Time	LOC	Time	LOC	Time	LOC	Time	LOC	Time
Vector Addition	17	-	1	+++	2	+++	3	+++	1	+++	15	-
Dot Product	19	- -	2	+++	3	+++	4	++	2	+	17	- -
Matrix Addition	18	-	1	+++	2	+++	3	+++	1	+++	15	-
Outer Product	17	-	1	+++	2	+++	3	+++	1	+++	15	-
Matrix-Vector Product	18	-	1	+++	2	+++	3	+++	1	+++	16	-
Matrix Transpose	17	- -	1	+++	2	+++	3	+++	1	+++	15	- - -
Matrix Product	18	- - -	1	+++	2	+++	3	++	1	+++	16	- - -
Gaussian Elimination	21	- - -	3	+	4	+	4	+	2	+	23	- - -
Jacobi Method	19	- - -	1	+	2	+	3	+	4	+	12	- - -

Table 4: Additional lines of code and development time in the multi-core programming models [Note: LOC means the number of lines of code and time means the development time]

implementations, a programmer can easily insert compiler directives (i.e., pragmas) or keywords into sequential code to tell the compiler which parts of the code should be executed in parallel. These models have the advantage that the only additions necessary to the sequential code are pragmas or keywords. In the SWARM implementation, a programmer uses constructs for parallelisation, e.g., to parallelise a *for* loop, the `par_do` construct should be used at the start of the loop; the `par_do` construct implicitly partitions the loop among the cores without the need for coordinating overheads such as synchronisation between cores. Furthermore, the SWARM framework provides the programmer with library functions and with calls to synchronisation and reduction operations. The syntax of these functions is easy to use because only a few arguments are required. In the TBB programming model, the programmer also uses template functions to parallelise a *for* loop and to perform reduction using the lambda function feature. Note that the lambda function is used for anonymous function definitions, meaning that a *for* loop could be passed to TBB's `parallel_for` function in-place, i.e., without defining a function object [44, 50]. This new feature is important because it helps reduce the programming effort related to function objects, for which many lines of code are required. Finally, as observed in the previous section, the OpenMP, SWARM, Cilk Plus, and TBB frameworks express data parallelism and reduction easily and directly because these frameworks provide ready-to-use parallel constructs or functions that do not require additional lines of code. This observation is supported by the fact that the number of additional lines of code required for these four programming models was smaller than for the others and the development time was sufficiently small. Finally, the code structure for the OpenMP, SWARM, Cilk Plus, and TBB frameworks was simple and easy to understand, and no complex programming effort was required.

The Pthreads implementations required more additional lines of code than their sequential counterparts, and the programming effort was complex. The Pthreads programming model provides a programmer with low-level library routines, and unlike the other frameworks, algorithm parallelisation is not automatic, meaning that the programmer is explicitly responsible for writing code for parallelisation, data distribution to each thread, and thread synchronisation. In other words, a manual code transformation is required to achieve parallelism. To parallelise a *for* loop, the programmer must create, join, and synchronise threads explicitly. Furthermore, the programmer must reorganise the code from the serial function into a thread function, and thread inputs must be grouped into a single data structure. Moreover, the code for the thread function must be modified so that each thread works on its own local data block. The result of all this is that Pthreads has the longest coding and development time of all the algorithms.

Finally, the FastFlow algorithm implementations required many additional lines of code, although C++ templates were provided. The code for the FastFlow programming model is easier to understand than the others, but because each algorithm was implemented as an object class, additional statements were required as well as the core code of the algorithm. In other words, this model required restructuring of the sequential code to make it more object-oriented. Therefore, this programming model requires significant programming effort and development time.

The programming effort or the development time required by each programming framework depends on the number of additional lines of code. In other words, the more additional lines of code required to parallelise a function, the longer the development time, and vice versa. Furthermore, there is a relationship between the support for parallel patterns provided by each framework and the expertise level of the programmer. In other words, the programming models that directly support parallel patterns through ready-to-use parallel constructs require little programming experience

Characteristic - Model	Pthreads	OpenMP	Cilk Plus	TBB	SWARM	FastFlow
Library popularity	High	High	Medium	Medium	Low	Low
Level of abstraction	Low	High	High	High	High	High
Programming style	C	C	C++	C++	C	C++
Help and documentation	Very good	Very good	Very good	Very Good	Good	Good
Learning curve	High	Low	Low	Medium	Low	Medium

Table 5: The characteristics of the multi-core programming models

from the user, whereas when parallel patterns are not supported directly by a model and they must be implemented with other features, advanced programming experience is required, as for the Pthreads and FastFlow models.

Regarding the other software engineering parameters, the Pthreads and OpenMP programming models are the most popular with programmers because they were developed earlier and have been used extensively by parallel computing researchers. The remaining programming models have medium or low popularity because they were developed when multi-core processors were introduced. The Pthreads framework has a low level of abstraction because the programmer is responsible for writing code for managing thread parallelism and synchronisation, whereas the other frameworks provide a high level of abstraction that can implement the same kind of thread parallelism using ready-to-use parallel constructs. These frameworks, except for Pthreads, provide an abstract programming model which enables the programmer to hide the details of thread programming, meaning that threads do not have to be managed directly as in Pthreads. Furthermore, high-level frameworks such as OpenMP, Cilk Plus, TBB, and SWARM provide similar parallel programming constructs to express the map and reduction patterns. As for programming style, parallel code in TBB, Cilk Plus, and FastFlow uses the C++ object-oriented style instead of the C procedural style used in the remaining frameworks. The code in TBB, Cilk Plus, and FastFlow is written in C++ and uses object-oriented principles and template functions. This has resulted in the initial sequential C code becoming a mixture of C and C++ code. Moreover, Pthreads, OpenMP, and Intel's two frameworks (Cilk Plus and TBB) support online documentation, manuals, books, and slides and provide many code examples. In particular, there are many very good teaching tutorials for the Pthreads and OpenMP frameworks. On the other hand, few manuals and code examples exist for the SWARM and FastFlow frameworks. Finally, the learning curve for the Pthreads framework is long for novice programmers because it requires low-level programming and because understanding the complex features and function syntax takes a long time, whereas frameworks such as OpenMP, Cilk Plus, TBB, and SWARM have short learning curves because programmers simply insert compiler directives or use available parallelisation constructs in the sequential code, meaning that these frameworks require the least amount of time (a few days) to learn the basic features of the model. The FastFlow framework has a moderately steep learning curve because programmers must study much terminology to understand the functionality of the object-oriented design. All these characteristics of the multi-core programming models are summarised in Table 5.

6. Conclusions

In this research, a set of linear algebra methods commonly used in the field of scientific computing has been parallelised using six multi-core programming models. Moreover, an extended quantitative and qualitative comparison of these programming models has been performed to answer the question: which is the appropriate programming model for implementing a scientific computing kernel on a multi-core system to achieve a balance between high programmer productivity and high performance? Based on this exploratory study, the following general conclusions can be drawn. First, some programming models have a single-core overhead (i.e., Pthreads, Cilk Plus, TBB, and FastFlow) compared to a sequential C program. However, this overhead has little effect on performance as the matrix size grows. Second, a significant reduction in execution time occurs as the number of cores increases when the amount of parallelisable work is high, such as for matrix addition, outer product, matrix-vector product, matrix product, matrix transpose, Gaussian elimination, and Jacobi operations. On the other hand, the performance of vector addition and dot product operations was poor because only a small amount of the work was parallelisable. Third, computationally intensive kernels, except for vector addition and dot product, had excellent scalability as a function of matrix size. Fourth, the best overall performance as a function of number of cores and of matrix size was shown by the SWARM

and OpenMP frameworks, followed by the Cilk Plus, TBB, and FastFlow models. Fifth, most high-level programming models such as OpenMP, Cilk Plus, TBB, and SWARM support parallel patterns through ready-to-use parallel constructs and require only a small number of additional or modified lines of code to be added to sequential code with minimal programming effort and a short learning curve. On the other hand, the Pthreads and FastFlow models do not support parallel patterns directly and require restructuring sequential code into parallel code with intensive programming effort. Finally, as a general conclusion, the OpenMP and SWARM programming models are suitable for parallelising computationally intensive methods and matrix operations of moderate size on multi-core platforms. They require little programming effort and give satisfactory performance as a function of number of cores and of matrix size. Furthermore, the FastFlow model can be used to implement large matrix operations such as matrix product with good performance, but this requires more programming effort. A programmer interested in implementing parallel methods with little programming effort for medium performance could use the Cilk Plus and TBB programming models.

Finally, a number of directions for future research have been identified. First, the quantitative and qualitative comparisons of programming models could be extended to other scientific computing algorithms such as matrix factorisation and similar applications from computational statistics and data analysis. Second, the present quantitative and qualitative evaluation could be extended to include advanced algorithmic techniques used for matrix operations such as the tiling technique, the divide and conquer strategy, and cache-oblivious algorithms. When implementing these algorithmic techniques, the performance of the parallel programming models could be compared with that of the Intel MKL BLAS library. Third, the performance comparison of programming models could be extended to sparse matrix computations with real-world sparse matrices from the Matrix Market repository [8]. Fourth, the scalability analysis, where the problem size and the number of cores were increased concurrently, could be extended to other matrix operations except for the outer and matrix products. Finally, a quantitative and rigorous analysis could be performed by gathering data on the development time of matrix operations for the various parallel programming models.

Acknowledgements

The authors would like to thank the anonymous reviewers for many helpful comments and suggestions, which have greatly improved the presentation of this paper.

Appendix

The tables can be retrieved from the `http://users.uom.gr/~pmichailidis/jpapers/appendix-anova-ci.pdf`

References

[1] Intel Cilk Plus, http://software.intel.com/en-us/articles/intel-cilk-plus/ (2012).
[2] Intel Threading Building Blocks, http://threadingbuildingblocks.org/ (2012).
[3] The openMP API specification for parallel programming, http://openmp.org/wp/ (2012).
[4] AMD microprocessors, http://en.wikipedia.org/wiki/List_of_AMD_microprocessors (2014).
[5] IBM Power8 architecture, http://en.wikipedia.org/wiki/POWER8 (2014).
[6] Intel Math Kernel Library - Documentation, http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation (2014).
[7] Intel Xeon, http://en.wikipedia.org/wiki/Xeon (2014).
[8] Matrix market repository, http://math.nist.gov/MatrixMarket/ (2014).
[9] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, S. Tomov, Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects, Journal of Physics: Conference Series 180 (1) (2009) 012037.
[10] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, M. Torquati, Accelerating code on multi-cores with FastFlow, in: Proceedings of the 17th International Conference on Parallel processing - Volume Part II, Euro-Par'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 170–181.
[11] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Torquati, Programming Multi-core and Many-core Computing Systems, chap. FastFlow: high-level and efficient streaming on multi-core, Wiley, 2011.
[12] M. Aldinucci, M. Meneghin, M. Torquati, Efficient Smith-Waterman on multi-core with FastFlow, in: Proceedings of 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2010, pp. 195 –199.
[13] A. Ali, U. Dastgeer, C. Kessler, OpenCL for programming shared memory multicore CPUs, in: Proceedings of MULTIPROG-2012 Workshop at HiPEAC-2012, 2012.

[14] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, D. Sorensen, LAPACK Users' Guide (Third Ed.), Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.
[15] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, G. Zhang, The design of OpenMP tasks, IEEE Trans. Parallel Distrib. Syst. 20 (3) (2009) 404–418.
[16] D. A. Bader, V. Kanade, K. Madduri, SWARM: A Parallel Programming Framework for Multicore Processors, in: Parallel and Distributed Processing Symposium, 2007. IPDPS 2007, 2007, pp. 1–8.
[17] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. V. der Vorst, Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition, SIAM, Philadelphia, PA, 1994.
[18] M. Birk, M. Zapf, M. Balzer, N. Ruiter, J. Becker, A comprehensive comparison of GPU- and FPGA-based acceleration of reflection image reconstruction for 3d ultrasound computer tomography, Journal of Real-Time Image Processing (2012) 1–12.
[19] H. Blaar, M. Legeler, T. Rauber, Efficiency of thread-parallel Java programs from scientific computing, in: Proceedings of the 16th International Parallel and Distributed Processing Symposium, IPDPS '02, IEEE Computer Society, Washington, DC, USA, 2002, pp. 131–.
[20] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley, ScaLAPACK User's Guide, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.
[21] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, Y. Zhou, Cilk: An efficient multithreaded runtime system, in: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Santa Barbara, California, 1995, pp. 207–216.
[22] J. M. Bull, Measuring Synchronisation and Scheduling Overheads in OpenMP, in: In Proceedings of First European Workshop on OpenMP, 1999, pp. 99–105.
[23] A. Buttari, J. Dongarra, P. Husbands, J. Kurzak, K. Yelick, Multithreading for synchronization tolerance in matrix factorization, in: Journal of Physics: Conference Series 78(1), 2007.
[24] A. Buttari, J. Langou, J. Kurzak, J. Dongarra, A class of parallel tiled linear algebra algorithms for multicore architectures, Parallel Comput. 35 (1) (2009) 38–53.
[25] D. Buttlar, J. Farrell, B. Nichols, PThreads Programming: A POSIX Standard for Better Multiprocessing, O'Reilly Media, 1996.
[26] K. Chellapilla, S. Puri, P. Simard, High Performance Convolutional Neural Networks for Document Processing, in: G. Lorette (ed.), Tenth International Workshop on Frontiers in Handwriting Recognition, Université de Rennes 1, Suvisoft, La Baule (France).
[27] T. Deepak Shekhar, K. Varaganti, R. Suresh, R. Garg, R. Ramamoorthy, Comparison of parallel programming models for multicore architectures, in: Proceedings of IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011, pp. 1675 –1682.
[28] J. W. Demmel, Applied Numerical Linear Algebra, Society for Industrial and Applied Mathematics, 1997.
[29] J. Diaz, C. Munoz-Caro, A. Nino, A survey of parallel programming models and tools in the multi and many-core era, IEEE Transactions on Parallel and Distributed Systems 23 (8) (2012) 1369–1386.
[30] M. Z. F. Cantonnet, Y. Yao, T. El-Ghazawi., Productivity analysis of the UPC language, in: 18th International Symposium on Parallel and Distributed Processing, 2004.
[31] K.-F. Faxén, Wool-A work stealing library, SIGARCH Comput. Archit. News 36 (5) (2009) 93–100.
[32] M. Frigo, C. E. Leiserson, H. Prokop, S. Ramachandran, Cache-oblivious algorithms, ACM Trans. Algorithms 8 (1) (2012) 4:1–4:22.
[33] M. Frigo, V. Strumpen, The cache complexity of multithreaded cache oblivious algorithms, Theory of Computing Systems 45 (2) (2009) 203–233.
[34] J. Gentle, Matrix Algebra: Theory, Computations, and Applications in Statistics, Springer Texts in Statistics, Springer, 2007.
[35] G. Golub, C. Van Loan, Matrix Computations, 2nd ed., Johns Hopkins University Press, 1989.
[36] J. Gomez-Pulido, M. Vega-Rodriguez, J. Sanchez-Perez, S. Priem-Mendes, V. Carreira, Accelerating floating-point fitness functions in evolutionary algorithms: A FPGA-CPU-GPU performance comparison, Genetic Programming and Evolvable Machines 12 (4) (2011) 403–427.
[37] K. Goto, R. A. v. d. Geijn, Anatomy of high-performance matrix multiplication, ACM Trans. Math. Softw. 34 (3) (2008) 12:1–12:25.
[38] A. Grama, G. Karypis, V. Kumar, A. Gupta, Introduction to Parallel Computing, Pearson Education, 2nd ed., Addison-Wesley, 2003.
[39] A. G. Gray, Fast Kernel Matrix-Vector Multiplication with Application to Gaussian Process Regression, Tech. rep., Carnegie Mellon University (2004).
[40] C. Grozea, Z. Bankovic, P. Laskov, FPGA vs. Multi-core CPUs vs. GPUs: Hands-on experience with a sorting application, in: R. Keller, D. Kramer, J.-P. Weiss (eds.), Facing the Multicore-Challenge, vol. 6310 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2010, pp. 105–117.
[41] J. Gunnels, J. Lee, S. Margulies, Efficient high-precision matrix algebra on parallel architectures for nonlinear combinatorial optimization, Mathematical Programming Computation 2 (2) (2010) 103–124.
[42] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, J. K. Hollingsworth, M. V. Zelkowitz, Parallel programmer productivity: A case study of novice parallel programmers, in: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, SC '05, IEEE Computer Society, Washington, DC, USA, 2005, pp. 35–.
[43] H. Kasim, V. March, R. Zhang, S. See, Survey on parallel programming model, in: Proceedings of the IFIP International Conference on Network and Parallel Computing, NPC '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 266–275.
[44] P. Kegel, M. Schellmann, S. Gorlatch, Comparing programming models for medical imaging on multi-core systems, Concurrency and Computation: Practice and Experience 23 (10) (2011) 1051–1065.
[45] S. Kesturt, J. Davis, O. Williams, BLAS comparison on FPGA, CPU and GPU, 2010, pp. 288–293.
[46] J. Kurzak, H. Ltaief, J. Dongarra, R. M. Badia, Scheduling dense linear algebra operations on multicore processors, Concurr. Comput. : Pract. Exper. 22 (1) (2010) 15–44.
[47] X. S. Li, Evaluation of superLU on multicore architectures, Journal of Physics: Conference Series 125 (1) (2008) 012079.
[48] M. Luff, Empirically investigating parallel programming paradigms: A null result, in: Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU), 2009.
[49] A. Marowka, Empirical analysis of parallelism overheads on CMPs, in: Proceedings of the 8th International Conference on Parallel processing

and applied mathematics: Part I, PPAM'09, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 596–605.
[50] M. McCool, J. Reinders, A. Robison, Structured Parallel Programming: Patterns for Efficient Computation, Elsevier Science, 2012.
[51] S. F. McGinn, R. E. Shaw, Parallel Gaussian elimination using OpenMP and MPI, in: Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications, HPCS '02, IEEE Computer Society, Washington, DC, USA, 2002, pp. 169–173.
[52] R. Membarth, F. Hannig, J. Teich, M. Krner, W. Eckert, Frameworks for multi-core architectures: A comprehensive evaluation using 2d/3d image registration, in: M. Berekovic, W. Fornaciari, U. Brinkschulte, C. Silvano (eds.), Architecture of Computing Systems - ARCS 2011, vol. 6566 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 62–73.
[53] P. D. Michailidis, K. G. Margaritis, Implementing parallel LU factorization with pipelining on a multicore using OpenMP, in: Proceedings of the 2010 13th IEEE International Conference on Computational Science and Engineering, CSE '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 253–260.
[54] P. D. Michailidis, K. G. Margaritis, Performance models for matrix computations on multicore processors using OpenMP, in: Proceedings of International Conference on Parallel and Distributed Computing Applications and Technologies, IEEE Computer Society, Los Alamitos, CA, USA, 2010, pp. 375–380.
[55] P. D. Michailidis, K. G. Margaritis, Open multi processing (OpenMP) of Gauss-Jordan method for solving system of linear equations, in: Proceedings of IEEE 11th International Conference on Computer and Information Technology (CIT), 2011, pp. 314 – 319.
[56] P. D. Michailidis, K. G. Margaritis, Computational comparison of some multi-core programming tools for basic matrix computations, in: Proceedings of 9th IEEE International Conference on Embedded Software and Systems (HPCC-ICESS), and IEEE 14th International Conference on High Performance Computing and Communication, 2012, pp. 143–150.
[57] P. D. Michailidis, K. G. Margaritis, Implementing basic computational kernels of linear algebra on multicore, in: Proceedings of 16th Panhellenic Conference on Informatics, IEEE Computer Society, Los Alamitos, CA, USA, 2012, pp. 217–222.
[58] P. D. Michailidis, K. G. Margaritis, Performance study of matrix computations using multi-core programming tools, in: M. Ivanovic, Z. Budimac, M. Radovanovic (eds.), Proceedings of 5th Balkan Conference in Informatics, ACM, 2012, pp. 186–192.
[59] A. Podobas, M. Brorsson, A comparison of some recent task-based parallel programming models, in: Proceedings of the 3rd Workshop on Programmability Issues for Multi-Core Computers, (MULTIPROG'2010), Jan 2010, Pisa, 2010.
[60] T. Rauber, G. Rünger, Parallel Programming - for Multicore and Cluster Systems, Springer, 2010.
[61] S. Ravela, Comparison of shared memory based parallel programming models, Master's thesis, Biekinge Institute of Technology (2010).
[62] J. Reinders, Intel Threading Building Blocks - Outfitting C++ for Multi-core Processor Parallelism, O'Reilly, 2007.
[63] G. Rünger, M. Schwind, Fast recursive matrix multiplication for multi-core architectures, Procedia CS 1 (1) (2010) 67–76.
[64] C. Sadowski, A. Shewmaker, The last mile: Parallel programming and usability, in: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, FoSER '10, ACM, New York, NY, USA, 2010, pp. 309–314.
[65] L. Sanchez, J. Fernandez, R. Sotomayor, J. Garcia, A comparative evaluation of parallel programming models for shared-memory architectures, in: Proceedings of IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA), 2012, pp. 363 –370.
[66] W.-H. Steeb, Y. Hardy, A. Hardy, R. Stoop, Problems & Solutions in Scientific Computing with C++ and Java Simulations, World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2004.
[67] D. Szafron, J. Schaeffer, An experiment to measure the usability of parallel programming systems, Concurrency: Practice and Experience 8 (2) (1996) 147–166.
[68] D. Tamir, N. Shaked, W. Geerts, S. Dolev, Parallel decomposition of combinatorial optimization problems using electro-optical vector by matrix multiplication architecture, The Journal of Supercomputing 62 (2) (2012) 633–655.
[69] R. A. van de Geijn, Using PLAPACK: Parallel Linear Algebra Package, MIT Press, Cambridge, MA, USA, 1997.
[70] M. M. Wolf, M. T. Heath, Combinatorial Optimization of Matrix-Vector Multiplication in Finite Element Assembly, SIAM Journal on Scientific Computing 31 (4) (2009) 2960–2980.
[71] D. Zou, Y. Dou, F. Xia, Optimization schemes and performance evaluation of smith-waterman algorithm on CPU, GPU and FPGA, Concurr. Comput. : Pract. Exper. 24 (14) (2012) 1625–1644.
[72] S. Zuckerman, M. Pérache, W. Jalby, Fine tuning matrix multiplications on multicore, in: Proceedings of the 15th international conference on High performance computing, HiPC'08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 30–41.