Sánchez, L.M., Fernández, J., Sotomayor, R., Escolar, S., García, J.D. (2013). A Comparative Study and Evaluation of Parallel Programming Models for Shared-Memory Parallel Architectures. *New Generation Computing*, Volume 31, Issue 3, pp 139–161

DOI: 10.1007/s00354-013-0301-5

A Comparative Study and Evaluation of Parallel Programming Models for Shared-Memory Parallel Architectures

Luis Miguel Sanchez, Javier Fernandez, Rafael Sotomayor, Soledad Escolar and J. Daniel Garcia

Computer Architecture and Technology Area.
UNIVERSIDAD CARLOS III DE MADRID.
Madrid, Colmenarejo 28270 SPAIN

Correspondence author: lmsan@arcos.inf.uc3m.es

Received 1 November 2012 [*1]

Abstract Nowadays, shared-memory parallel architectures have evolved and new programming frameworks have appeared that exploit these architectures: OpenMP, TBB, Cilk Plus, ArBB and OpenCL. This article focuses on the most extended of these frameworks in commercial and scientific areas. This paper shows a comparative study of these frameworks and an evaluation. The study covers several capacities, such as task deployment, scheduling techniques, or programming language abstractions. The evaluation measures three dimensions: code development complexity, performance and efficiency, measure as speedup per watt. For this evaluation, several parallel benchmarks have been implemented with each framework. These benchmarks are created to cover certain scenarios, like regular memory access or irregular computation. The conclusions show some highlights, like the fact that some frameworks (OpenMP, Cilk Plus) are better for transforming quickly a sequential code, others

[*1] The final version of this paper is available at link.springer.com. Cite as: A Comparative Study and Evaluation of Parallel Programming Models for Shared-Memory Parallel Architectures. Luis Miguel Snchez, Rafael Sotomayor, J. Daniel Garca, Javier Fernndez, Soledad Escolar. New Generation Computing, 31(3):139-161. 07/2013. ISSN: 0288-3635. DOI:10.1007/s00354-013-0301-5

(TBB) have a small footprint which is ideal for small problems, and others (OpenCL) are suited for heterogeneous architectures but they require a very complex development process. The conclusions also show that the vectorization support is more critical than multitasking to achieve efficiency for those problems where this approach fits.

Keywords Parallel programming, vector instructions, multithreading, performance analysis, efficiency analysis, power consumption

§1 Introduction

Currently, commercial off-the-shelf computers include several hardware features that impact on performance of user applications making use of parallel programming paradigms. Those characteristics include features to exploit applications parallelism. They can be classified as: Instruction-Level Parallelism (ILP), Thread-Level Parallelism (TLP) (e.g. hyper-threading, multicore) and Data-Level Parallelism (DLP) (e.g. CPU vector instructions and other SIMD architectures). Only the ILP approach can be effectively hidden from the application software. In contrast, TLP and DLP have the inherent problem of how to develop parallel applications that exploit these features to optimize performance.

In order to address the former problem, several parallel programming frameworks have appeared to ease the development of this kind of applications. Among the goals of these frameworks are to ease the implementation and to hide from the programmer the low-level details of the parallel hardware features. This paper shows an evaluation of the following frameworks: OpenMP, Intel TBB, Intel Cilk plus, Intel ArBB and OpenCL. These frameworks have been chosen because each one is focused on different hardware features and different approaches to exploit them. Even though most of these frameworks are valid for different architectures and CPUs, their performance may experience great variations on different hardware architectures. However, the programming framework is not the only software responsible for the final performance. The compiler has also a great influence on the performance.

This paper is organized as follows: Section 2 shows the related work. Section 3 describes several features for parallel computation that are present in modern hardware. Section 4 enumerates parallel programming frameworks we have used in our study and classifies their main characteristics. Section 5, lists the set of benchmarks used for evaluation and describes the parallelism issues

they cover. Section 6 describes the evaluation procedure, the results we have obtained, and outlines our main conclusions. Finally Section 7 presents the final conclusions.

§2 Parallel Computing Architectures

Modern off-the-shelf computers include many hardware features which are suited for parallel computation. This section shows a brief summary of these features and the impact they can have on the performance and the development of parallel applications.

2.1 Thread-Level Parallelism (TLP)

Thread Level Parallelism (TLP) is the most extended trend used to take advantage of parallel hardware features. TLP creates the parallelism by executing concurrently several threads that coordinate their actions to achieve a common goal. Modern computers architectures include many features to improve TLP: Hyper-threading, several threads executing on one core; Multicores, multiple cores connected to the same memory banks; and ccNUMA architectures (Cache Coherent Non-Uniform Memory Access), multiple cores with their own memory banks but with a unified vision of the memory.

These features help to increase the performance if the application is split into several tasks. Every task is typically executed by a single thread, and all the threads are executed in parallel. The main challenge for the programmer is actually how to split the application into tasks efficiently, in such a way that they can exhibit a high degree of concurrency. The tasks must be equally weighted in terms of computation and time to obtain the maximum performance. Moreover, the programmer has to deal with the issues derived from managing data shared between tasks in a coordinated way. Usually, this involves a memory access policy that allows exclusive accesses for a task. A desirable trait for a parallel programming framework is that it eases this job, while obtaining the most efficient code possible. A parallel programming framework should simplify the process of writing parallel programs from scratch, and the process of modifying existing sequential programs into parallel ones.

2.2 Data-Level Parallelism (DLP)

Data-Level parallelism exploits parallelism by executing the same operation over each one of the elements of a big set of data in parallel. This is known

as the Single Instruction Multiple Data (SIMD) approach. Modern computers include many features to improve DLP, of which two most prominent approaches are: 1) to enhance the CPU with data parallel units that are used through special instructions (e.g. vector instructions); and 2) to use computing elements outside the CPU that are focused on handle SIMD code (e.g. GPUs).

Both approaches have advantages and disadvantages. The CPU can only execute a reduced number of simultaneous operations (in the order of tens), while the GPU can execute a larger number of operations (in the order of thousands). In contrast, the CPU uses the main memory directly, while the GPU has to transfer first the data to its local memory.

An application designed for SIMD architecture has to pack together the operations that are repeated over each element of a dataset using the appropriate instructions. Parallel programming frameworks are designed to ease the process of packing operations to the programmer. Specifically, some parallel programming frameworks oriented to CPU, with the support of the compiler, can reorder the code of the executable program in order to take advantage of the CPU vector instructions. This can be achieved without explicitly modifying the source code. Some parallel programming frameworks can work with CPUs and GPUs from different vendors. There are frameworks, such as OpenCL that can work with both without changing the source code.

2.3 Parallel Memory Access

Parallel programming frameworks also have to deal with issues related to memory access. As part of the memory hierarchy, caches have to ensure coherency when several cores write the same address by invalidating or updating the individual copies of data. Also, ccNUMA architectures penalize the performance when a core accesses a memory bank outside those that belong to this core. This penalization becomes more pronounced when tasks are moved between cores. In the case of GPUs, the lack of a unified CPU-GPU memory space forces the program to transfer data from one to another. Moreover, the GPU memory organization has several memory spaces that may be shared between the processing units. This makes codifying applications very difficult. Parallel programming frameworks have to manage these memory related problems using techniques as the ones mentioned below:

- Defining special data structures to act as data containers. This improves the memory access to those datasets that follow regular access pattern.

- Rearranging the executable code to improve the memory access.
- Offering task scheduling policies that consider the cache and the ccNUMA effects when moving tasks between cores.
- Using synchronization mechanisms only when necessary to enhance memory access.

2.4 Energy Consumption vs. Performance in Parallel Architectures

The use of TLP and DLP paradigms allows to increase the performance of the applications. However, it generally increases energy consumption. Nevertheless, it is not easy to find a correlation between both. Energy consumption depends on several factors, such as the hardware components which have different consumption rates, the time of usage of a resource (e.g. memory, CPU, GPU, etc.), etc. For example, there are differences between using several cores or one core with a vector processing unit. Performance increases when applications make use of the underlying hardware in an efficient way. This means to execute applications faster or doing more jobs in the same interval of time using the same hardware. Hardware components are used efficiently only if the increase in performance is at least equivalent to the increase in energy consumption.

Parallel programming frameworks are only focused on increasing performance. However, nowadays the economic cost of the energy consumption is becoming a huge part of the overall cost of the system. Therefore, the additional cost in energy consumption may not compensate the increase in performance. Consequently, it is a good practice to study applications efficiency together with the raw performance. Here, efficiency is measured as performance per energy unit [11].

§3 Parallel Programming Frameworks

This section presents the parallel frameworks evaluated in this paper, which are the following ones:

Open Multi-Processing (OpenMP) [4] is an open specification that defines a language extension for task parallelization based on compiler directives. OpenMP includes parallel loops, parallel regions that are executed by all the cores, and support for shared variables, among others.

Intel Threading Building Blocks (TBB) [12] is a C++ library for task parallelization. Its features include classes that implement generic tasks,

parallel loops, task scheduling, etc.

Intel Cilk Plus [19] is an extension to the C/C++ languages for task parallelization, including spawn functions, parallel loops and special arrays that ease the vectorization.

Intel Array Building Blocks (ArBB) [5] is a C++ library that allows defining small fragments of code called kernels which are parallelized and used vector instructions. ArBB includes a specific set of basic data classes and methods to define these kernels, which are executed using an abstract machine that includes just-in-time compilation and optimization, separated memory management and a task scheduler that adapts to the architecture without recompiling the code.

Open Computing Language (OpenCL) [17] is a standard extension of C/C++. It allows to write SIMD kernels that can be compiled just-in-time and executed on both CPUs and GPUs without recompiling the code. It can also generate specific binary files for each specific architecture.

Intel Math Kernel Library (MKL) [8] is a library of optimized mathematical functions using multitasks and CPU vectorization.

These frameworks use different approaches to implement parallel applications and to use the hardware efficiently. Thus, each framework has its own characteristics in terms of task deployment, vectorization support, task scheduling, programming language abstractions, and configuration capabilities. In this section we compare those characteristics across the considered frameworks. Table 1 shows a summary of the characteristics. Of these, only MKL is not a general purpose library, rather a mathematical library. Therefore it is not included in the comparison.

3.1 Task deployment

Each framework deploys the tasks over the processing units according to different approaches. OpenMP, TBB and Cilk Plus allow the programmer to define individual tasks for each core, or to define the whole problem as an iterative loop and let the framework to unroll the loop using several tasks. These frameworks offer two methods to obtain the tasks from a loop and to schedule them: 1) a static method and 2) a dynamic method. The static method obtains the tasks dividing the number of iterations between the number of cores. Thus, there are as many tasks as cores, and they are scheduled from the beginning. The dynamic method assigns a predefined, small number of iterations to each

task. Therefore, the number of tasks can be bigger than the number of cores. After that, new tasks are created dynamically during the whole execution and distributed between idle cores. OpenMP can choose between a static or dynamic approach. TBB and Cilk Plus implement a dynamic approach OpenCL and ArBB require the use of kernels, functions of code that are executed over different data. Both deploy a copy of the same kernel onto the processing unit of the GPUs and CPUs. Then, each unit executes the kernel over different data. OpenCL requires that the programmer decides the data size handle by one kernel. Also the kernel in OpenCL is responsible for locating and managing the data assigned. In contrast, ArBB distributes, locates and manages the data for each kernel transparently to the kernel and the programmer. Internally, ArBB implements the kernels as tasks using the same approach that TBB and Cilk Plus use for parallel loops.

3.2 Vectorization support

Each framework uses different techniques to adapt the application to the vectorization support of each CPU. This adaptation is usually done by the compiler used for the target CPU. This approach, which is the one followed by OpenMP, TBB and Cilk Plus, requires to recompile the source code to use the vectorization support of a different CPU. ArBB compiles the kernels using its own compiler that generates an intermediate code. This code is interpreted on-the-fly for the target platform. Therefore, ArBB can use the vectorization support of the target CPU without recompiling the source code. OpenCL uses an heterogeneous approach that works for both GPUs and CPUs. The framework compiles the source code of the kernels using its own compiler. This compilation can be done on-the-fly. Another option is compile several version of the kernel and then the framework chooses one to be executed on-the-fly.

3.3 Task scheduling

Each framework implements its own strategy to schedule tasks onto processing units. OpenCL uses a low-level approach where one kernel is deployed onto several processing units and this deployment cannot change until all the units have finished. The kernel is responsible to know which processing unit and which data have been assigned.

The rest of the frameworks use a high-level approach. They implement software tasks that are executed on top of system threads. Each core has assigned

one system thread. The software tasks can be moved from one system thread to another. A scheduler decides which task is executed in every system thread and which ones are queued. OpenMP uses a global queue for all the system. TBB, Cilk Plus and ArBB use a local queue for each core and allow idle cores to steal tasks from other queues [7].

3.4 Programming language abstractions

Each framework includes different programming language abstractions to ease the implementation of the code. The level of these abstractions is directly related with the level of complexity of the final code. OpenCL uses low-level programming language abstractions that make the programmer responsible for: 1) controlling data transfers within the CPU and GPU memories 2) deploying the kernels over the computing units, and 3) distributing the data between the kernels.

The rest of the frameworks offer several high-level Programming language abstractions that ease the implementation and the adaptation to different CPUs. TBB framework includes the definition of user-defined task classes that can be instantiate as task objects executed in parallel. TBB also offers predefined methods that can dynamically instantiate and execute task objects of a certain class to implement different parallel abstractions (parallel loop, fork-join, pipeline, etc). These methods also distribute the data onto the task objects.

Cilk Plus framework can execute a function in a parallel task (spawn) and synchronize the result with the code of the main task. Cilk Plus can also define parallel loops using a keyword of the language (cilk_for). This framework also has an array notation that can define simple algebraic operations between each corresponding element of the two matrices. Finally, Cilk Plus incorporates an abstraction called hyperobject that allows the programmer to implement a reduction operation with a single object.

ArBB framework let the programmer to define kernels. The kernels are deployed by mapping them onto the data. This framework also has an array notation that can use any kernel as the operation between the two matrices.

OpenMP framework uses pragma directives to define the language abstractions. The main pragma directive is the parallel region that allow the user to define parallel loops, individual tasks, etc.

3.5 Configuration capabilities

Each framework offers different possibilities to configure the management of the tasks and the parallel programming abstractions. OpenMP framework allows modifying the management of the parallel loops using three methods: 1) the static method, 2) the dynamic method and 3) the guided method. The static method divided the loop iterations between the tasks. The dynamic and guided method delivers a fixed number of iterations to each task. This value is static with the dynamic method while it can be changed with the guided method. Cilk Plus framework allows to make changes to the size of the data that is delivered to each task and the maximum number of cores. TBB framework also allows to change the data size and the maximum number of cores. Furthermore, this framework can include a user-defined scheduling algorithm to replace the one by default. ArBB allows to change the maximum number of cores but it handles all the rest of the options. Finally OpenCL is the framework that is most sensitive to changes in the configuration. The reason is the lack of a scheduler and a default distribution of the data. This makes the performance of the application very sensitive to the data size that the programmer selects for each client.

	OMP	Cilk Plus	TBB	ArBB	OpenCL
Task Deployment	Tasks & Par. loops (stat/dyn)	Tasks & Par. loops (dynamic)	Tasks & Par. loops (dynamic)	SIMD kernels (dynamic)	SIMD kernels (static)
Vectorization support	Compiler	Compiler	Compiler	Comp. on-the-fly	Comp. on-the-fly & versions
Scheduling Techniques	Softw. tasks Glob. queue	Softw. tasks Local queue	Softw. tasks Local queue	Softw. tasks Local queue	Static
Language Abstractions	Parallel regions	*spawn* *cilk_for* Hyperobjects Array op.	Task obj. Par. methods	Kernels Array op.	Kernels
Configuration Capabilities	Data size Max. cores Scheduler	Data size Max. cores	Data size Max. cores Scheduler	Max. cores	Data size

Table 1 Summary of the Characteristics of the Parallel Frameworks.

§4 Benchmarks

We have evaluated the frameworks listed in Section 3 by using a set of benchmarks. These benchmarks cover some common scenarios representing complex problems, which are solved by decomposing them in parallel tasks. All of the frameworks proposed are designed to ease the programming of these

kinds of problems. Among the possible scenarios, we have left out, those that involves irregular data access, where the data is indexed in memory with an access pattern that is unknown at compilation time. We do not include them in this study because these scenarios are so hard and complex that require a most extensive study.

Next, we present the scenarios we have chosen to test the frameworks and the associated benchmarks:

Regular computation: This scenario involves a heavy computation with very little or none access to memory. This scenario involves computing a set of computational operations that generate partial results. These operations can be mapped onto cores by using different criteria. The key of this scenario is that all computation operations require the same computational effort.

To cover this scenario, we implement a benchmark to obtain the value of π. The benchmark is based on the numerical integration of:

$$\int_0^1 \frac{4}{1+x^2} dx$$

The integral becomes in a sum where the individual sums are grouped in several tasks. The partial results are accumulated together to compute the final result.

Irregular computation: In this scenario, each computation element requires a different amount of computational effort, which ranges from almost none to very significative, as opposite to the regular computation scenario.

We implement a benchmark to draw the Mandelbrot fractal. This benchmark performs iterative operations to calculate each element (pixel) of the image. The algorithm performs several iterations for computing each partial result. The number of iterations range between one to a certain value, which is different for each partial result. This means that some elements only iterate once and other may perform the maximum possible number of iterations. The challenge of the mapping between computational operations to cores is to achieve that all the cores perform the same computational load.

Regular memory access: This scenario involves some computation and a great number of memory accesses. The problem requires to compute several partial results where each one requires a large number of memory accesses. The key is that the data location is known at compilation time and it normally follows a regular pattern.

We implement two benchmarks to illustrate this scenario: 1) the matrix multiplication algorithm and 2) the 2D convolution algorithm, a filtering tech-

nique used to apply effects to images. Both of them make an intensive use of matrix-based operations. The matrix multiplication involves access to one row and to one column to obtain one element. The 2D convolution algorithm requires access all the neighbors elements to an specific element on the image. The matrix multiplication benchmark is implemented using the framework-specific extensions for matrixes. In contrast, the 2D convolution is implemented using only iteration loops over the matrix elements.

Data reduction: This scenario involves a large number of shared data modifications. In this scenario, the tasks compute their partial results and write them on a shared memory area. The final result is computed by a task, which performs a reduction operation over all of the partial results. The more complex the results are, the more difficult the reduction is.

We implement a benchmark to calculate the histogram of an image, which is the number of pixels of each colour that are present in the image. We split the image into fragments that are processed by different tasks. Each task creates an array to store the occurrences of grey shades on the fragment. At the end, all the arrays created are accumulated into one common array. For this purpose, the operations for data reduction of each framework are used.

§5 Evaluation

We evaluate the parallel programming frameworks by executing and comparing the benchmarks implemented (described in Section 4) under the next metrics: the complexity of the code, performance and energy.

5.1 Code development complexity

For evaluating the code development complexity, we use the Code Churn estimation [10], which allows to measure the code transformation complexity based on the amount of added (LA), deleted (LD) and modified (LM) lines. We want to see the number of transformations done in the parallel version with regard to the sequential version. Table 2 shows the results (LA, LD, LM) for each benchmark and for each parallel programming framework. We also show the total churn value computed as $LA + LD + LM$.

The results show that OpenCL is the framework with the highest complexity in terms of code development. It is important to notice that the implementation performed is as general as possible and it has not been manually tuned for the target architecture. A manual tuning could probably improve the

		OMP		Cilk Plus		TBB		ArBB		OpenCL	
MM	LA	3	**3**	7	**13**	23	**31**	23	**31**	119	**127**
	LD	0		5		8		8		8	
	LM	0		1		0		0		0	
PI	LA	4	**8**	7	**10**	20	**27**	17	**24**	127	**134**
	LD	3		1		7		7		7	
	LM	1		2		0		0		0	
Histogram	LA	12	**16**	13	**18**	28	**31**	11	**20**	222	**230**
	LD	0		0		2		8		8	
	LM	4		5		1		1		0	
Conv2D	LA	2	**2**	3	**4**	23	**37**	25	**39**	125	**139**
	LD	0		0		14		14		14	
	LM	0		1		0		0		0	
Mandelbrot	LA	3	**3**	3	**4**	32	**48**	27	**43**	247	**263**
	LD	0		0		16		16		16	
	LM	0		1		0		0		0	
Total		**32**		**49**		**174**		**157**		**893**	

Table 2 Churn code evaluation. Notation: MM: Matrix Multiplication; PI: PI benchmark; Histogram: Histogram benchmark; Conv2D: 2D Convolution benchmark; Mandelbrot: Mandelbrot benchmark.

performance observed but it will also increase greatly the complexity of the development. However, the results are still far higher than the others, with implies that OpenCL is in a different level of complexity than the rest. Next in complexity are the TBB and ArBB frameworks. Their results are close to each other, but much lower than OpenCL. Finally Cilk Plus and, specially, OpenMP show the lowest values, making them the frameworks of choice to port a sequential application with little modifications.

5.2 Performance evaluation

We evaluate each parallel programming framework by executing the benchmarks proposed in Section 4 on a multisocket multicore architecture with a ccNUMA architecture. It includes four Intel Core Xeon E7-4807 sockets. Each socket counts with 6 cores at 1.87 GHz with hyper-threading (two threads per core). This results in a total of 24 cores and 48 threads. Each CPU has a local memory of 32 GB in 4 banks with a total memory 128 GB. Also, each core includes the SSE4.2 instruction set of 128 bits.

We compile all the benchmarks using the Intel C/C++ compiler with two options: 1) '-O0', no optimization; and 2) '-O3', full optimization with vectorization, loop unrolling, etc. We measure the energy consumption and performance for these benchmarks. The benchmarks have been executed several

times, where in each execution we vary the amount of data to process and the number of parallel threads. Table 3 shows the version of the software used in this comparison, most of them are included in Intel Parallel Studio XE 2013.

Software	Intel compiler	Intel TBB	MKL	ArBB	Intel OpenCL driver
Version	13.0.0	4.1	11.0	1.0.0.030	1.5-15294

Table 3 Software version used.

Figures 1 and 2 show the performance results. Figures 1 depicts the performance results for the execution of each benchmark implemented in the corresponding framework using both no optimization (-O0) and full optimization (-O3). In the x-axis we represent different problem sizes. For π benchmark we use scientific notation to express the maximum resolution size. For the rest of the tests we present the problem size as the number of elements N in a square matrix of $N \times N$. The y-axis shows in a logarithmic scale the execution time in seconds for each test.

Figure 3 shows the performance varying the number of threads ranging between 2 and 48 while we maintain the largest problem size for each benchmark.

The results obtained may be summarized as follows:

Performance overload: Each framework has an overload due to the initial setup. This overload is mainly the result of the task deployment and task scheduling techniques used in each framework, and also the result of using the run-time that performs the on-line compilation in those frameworks that use this technique (ArBB and OpenCL). The overload includes threads creation, data distribution, and other actions. Other actions, like task scheduling, also has great initial overload, but they also create an overload during the whole execution that is lower than the initial one. The initial overload can be compensated if the problem size is large enough. However, for small size problems it may represent a high percentage of the total overload. Thus, small workloads may show a lower performance than the sequential version because this initial overload. Figure 1 shows this effect for benchmarks with a small problem size: for a resolution of 1e4 in the π benchmark or for images with size smaller or equal than 2048×2048 in case of histogram benchmark. The overload of the two sequential versions are represented by a flat line with crosses (-O0) and by a discontinued line with the symbol $\times$ (-O3). If the performance line of a benchmark is under these lines indicates that the framework outperforms the sequential versions. Otherwise, if the performance line is over them indicates the opposite.

The frameworks with the largest overload are OpenCL and ArBB due

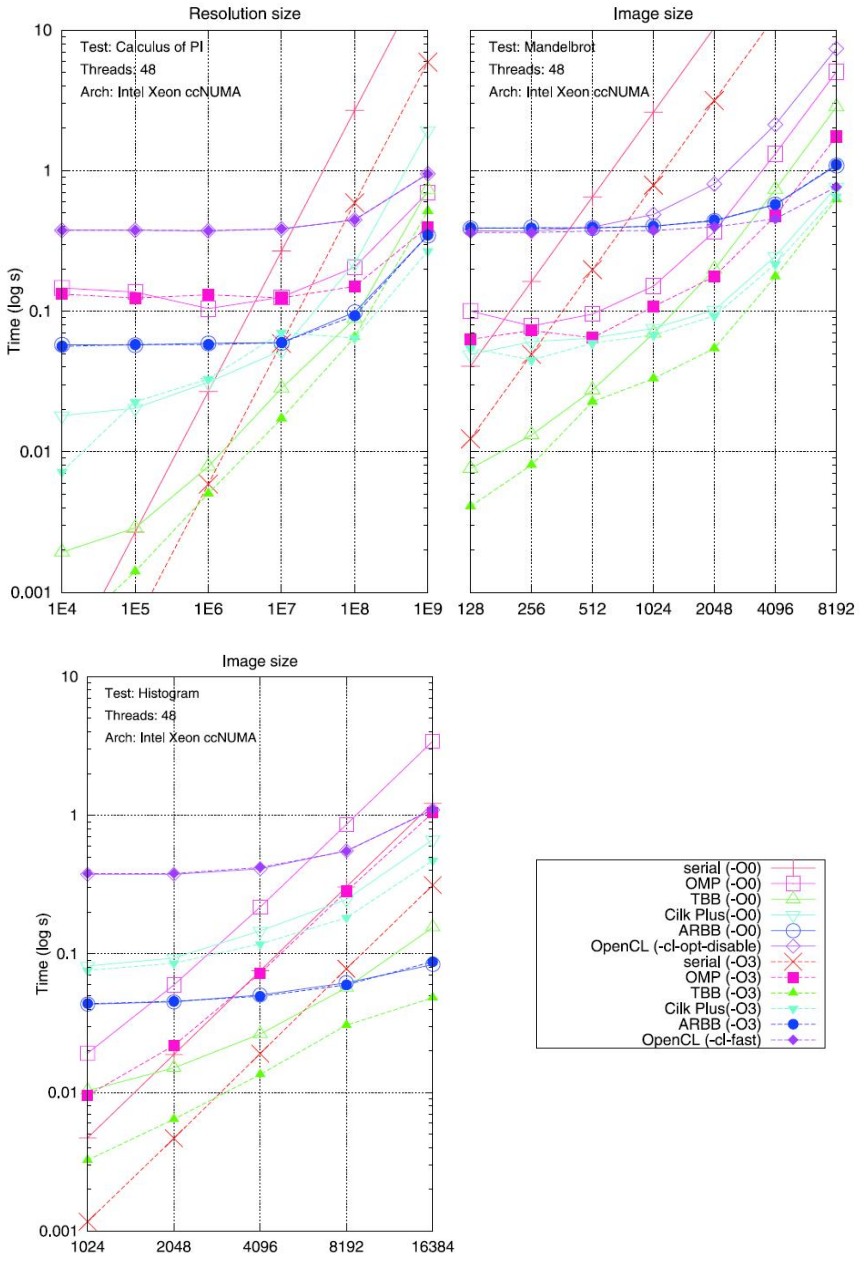

Fig. 1 Performance varying the problem size for Pi, Mandelbrot and Histogram benchmarks

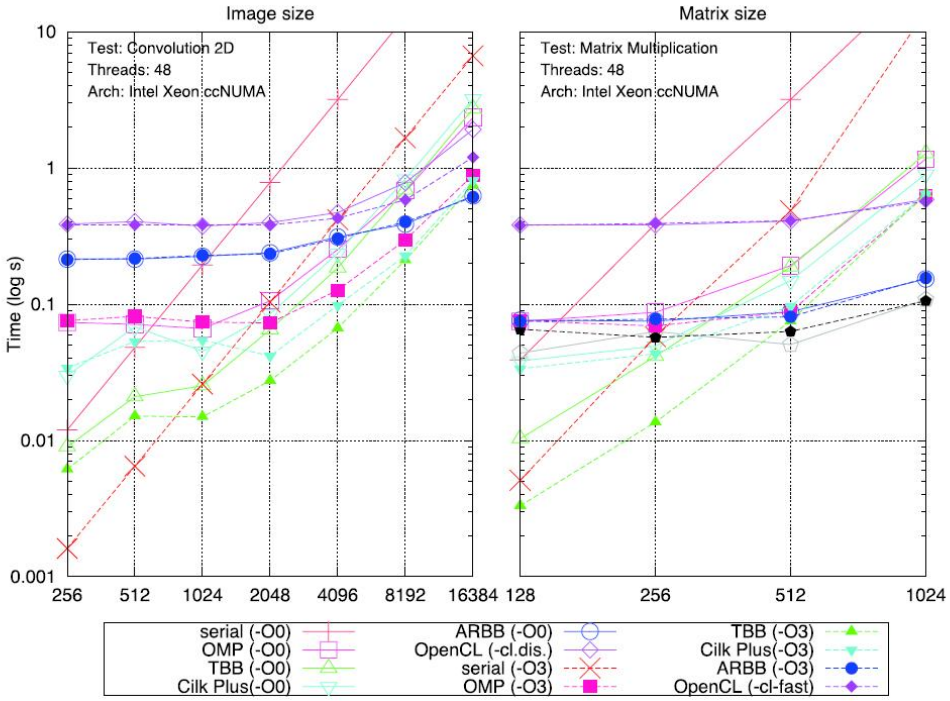

Fig. 2 Performance varying the problem size for Matrix Multiplication and Convolution 2D benchmarks

to its runtime, as shown in Figure 1 and 2 where their performance is lower than the sequential version for small and medium size problem. However, for large size problems, OpenCL and especially ArBB obtain better performance than other frameworks because they take more advantage of the vectorization support. This means that these frameworks may be used for large computation loads. A special case is the histogram benchmark, where the sequential version has a high hit rate for the cache. The parallel versions of that benchmark however are not able to achieve that rate and for this reason, they obtain a lower performance in most of the cases. The lighter frameworks are Cilk Plus and, especially TBB, which obtain better performance than others frameworks with smaller workloads because they impose a smaller overload in the initial setup and a better distribution of the data thanks to the dynamic scheduling over ccNUMA architectures.

As the framework overload becomes negligible, which occurs with larger workloads and when vectorization techniques are applied, the performance of most of the frameworks tends to converge.

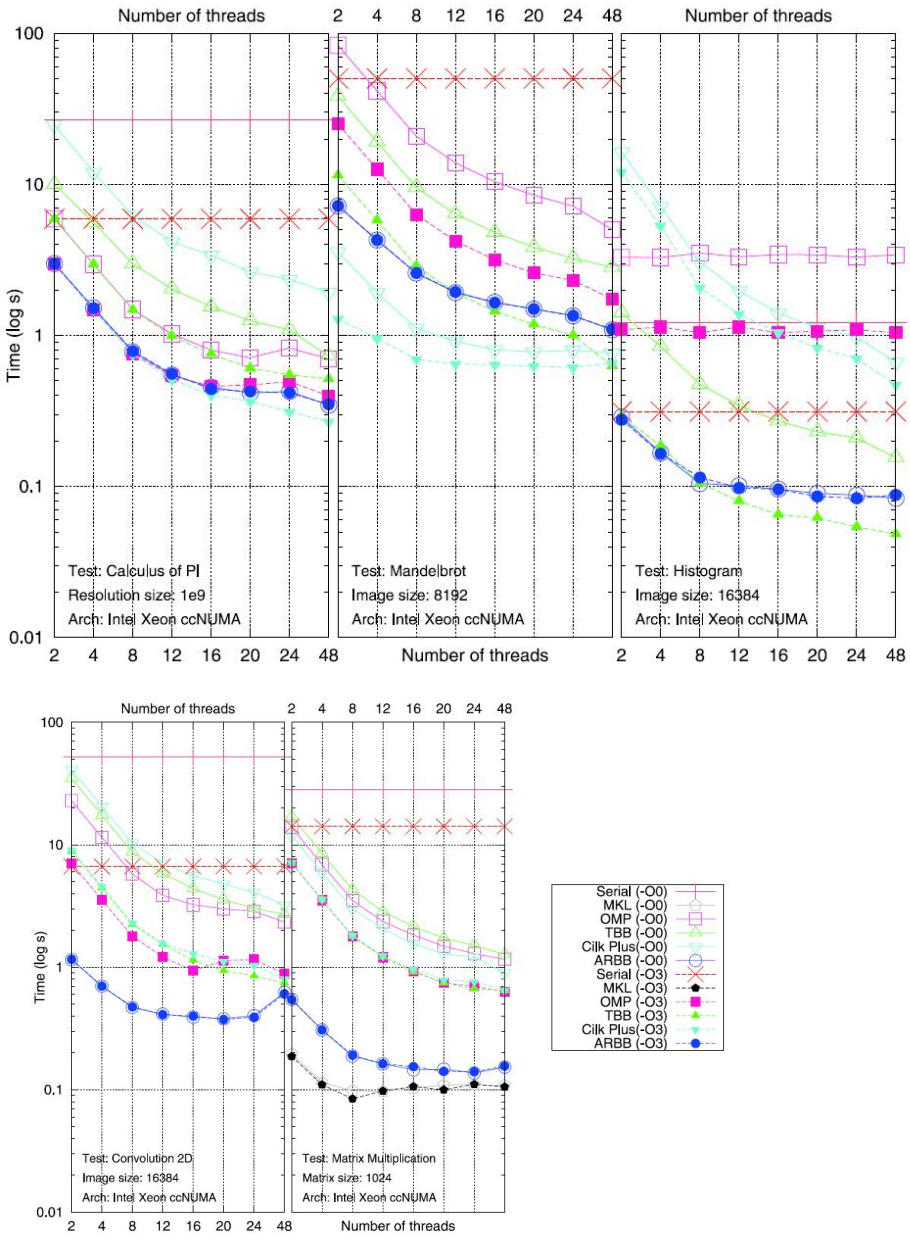

Fig. 3 Performance varying the number of threads

Vectorization: Some frameworks rely on the compiler to generate vector instructions for executables; the others generate intermediate code and, at execution time, they interpret this code. OpenMP, TBB and Cilk Plus follow the first trend. Figures 1, 2 and 3 show that, without loss of generality, the optimized version (-O3) of the benchmarks perform better than the non-optimized versions (-O0) for all problem sizes considered. OpenCL uses its own compiler for the kernels, so the options of the compiler have no effect. OpenCL benchmarks can be compiled just-in-time (option Comp in the legend) or precompiled (option Bin in the legend). The difference in terms of performance between both is only noticeable with large workloads and only for some benchmarks (Mandelbrot and Convolution 2D). ArBB and MKL obtain a similar performance without the compiler optimizations. The reason is that MKL is a library that is already compiled and ArBB uses its own compiler to generate an intermediate code that is converted into vectorized code at execution time. In fact, both OpenCL and ArBB generate the vectorized code at execution time, while OpenMP, Cilk Plus and TBB generate vectorized code at compilation time.

Performance of the reduction operations: The histogram benchmark in Figures 1, 2, and 3 shows that each framework performs differently due to the usage of specific reduction operations. The two frameworks that perform better than the sequential solution are TBB (using the *join* method) and ArBB (using matrix-specific reduce operations). In contrast, Cilk Plus hyperobjects, which implement their own reduction operations, perform worse than TBB and ArBB. OpenCL shares a global array which must be accessed via atomic functions, impacting negatively on the performance. OpenMP forces the programmer to define a hand-made parallel reduction phase, leaving to the programmer experience such a responsibility, which may impact on the applications performance.

The rest of the results in Figures 1, and 2 and 3 can be broken down for the set of benchmarks as follows:

- π benchmark involves a regular computation and a data reduction operation using simple data types, that is similar to the operation used in Histogram benchmark but with complex data types. The OpenMP code for the histogram is different because OpenMP cannot reduce complex data types. The performance of TBB using the *join* operation for the reduction of simple data types is worse than the reduction strategies of the rest of the frameworks, with the exception of OpenCL. However, the

same *join* operation has a great performance when it is employed with complex data types, as shown by the Histogram results.

- Mandelbrot benchmark involves irregular computation, which is a problem for the scheduling policies. OpenMP, with its dynamic scheduling, presents poorer results than the rest of the frameworks, which share the same scheduling policy. Furthermore, Cilk Plus shows remarkable results for all configurations, because its matrix-specific operations are used to operate several elements at once, resulting in a better vectorization.
- For 2D convolution and for matrix multiplication benchmarks TBB, Cilk Plus and OpenMP perform very similarly. ArBB, on the other hand, performs a little better on the convolution and a lot better on the matrix multiplication thanks to the array operations with generic kernels and a better vectorization. MKL, which can only be used for the matrix multiplication, obtains the best results.

5.3 Energy efficiency

We evaluate the energy consumption of the benchmarks proposed in Section 4 on an Intel Core i7-2600 socket that contains four cores at 3.40 GHz with hyper-threading (two threads per core) and 8 GB of memory. This is an Intel Sandy Bridge that includes the AVX vector instruction set of 256 bits. It has the ability of measuring energy consumption using the Likwid [13] tools. The energy consumption is also measured using an ammeter connected to each one of the power cables that feed the motherboard. The results are collected using an analog interface equipment from National Instruments, which is managed using an application made with LabView. The values obtained from both the Likwid tools and the ammeter confirms the same results for the energy consumption.

Figure 4 shows the efficiency results for all benchmarks described in Section 4. We measure efficiency as speedup per watt on y-axis, meanwhile in x-axis we show the different configurations for these benchmarks. All the benchmarks use the largest problem size. Also, Figure 4 shows the efficiency obtained for each benchmark by using different number of threads, ranging between 1 and 8, and using the non-optimization and full-optimization compiler options.

The results show that in most of cases, using one thread with vectorization is more efficient than using eight threads without vectorization. For example, consider the sequential solution of the 2D convolution benchmark, with is vectorized by the compiler. This configuration is more efficient than most of

the others, except ArBB with 8 threads. This is because ArBB always uses the vectorization support. The conclusion is that vectorization is the more efficient solution when the problem is suited to employ it. The reasons for this behaviour are that vectorization supposes that the problem meets several requirements like absence of synchronization, regular access to the memory and usage of shared control. Problems that do not meet these requirements cannot improve their performance using vectorization and have to rely on the TLP approach.

We present the results of power consumption using the ammeter for two benchmarks: π and Matrix Multiplication. We measure the power consumption of these benchmarks across the execution time.

First, Figure 5 shows π benchmark. The x-axis represents the execution time interval ranging between 176 and 225 seconds. The y-axis represents the power consumption in watts of two different hardware components: CPU is shown in a red line and memory is shown in a blue line. The graph shows eight executions of π: four were compiled with no optimization and the rest with full optimization. We indicate the number of threads used for each execution, ranging between 1 to 8.

Second, Figure 6 shows an execution of Matrix Multiplication benchmark using ArBB, MKL and OMP, all compiled with the full optimization option. We use different number of threads ranging between 1 to 8. The x-axis represents the execution time interval ranging between 10.5 and 44 seconds. The y-axis represents the power consumption in watts of CPU and RAM.

The results are summarized as follows:

- The execution time decreases when the number of threads grows.
- The power consumption increases with the number of threads. Also, if we compare the tests with 4 threads (without hyperthreading) and the tests with 8 threads (with hyperthreading) we see that the use of hyperthreading to increase the number of threads consumes a lot less than using new cores.
- Finally, the same test with vectorization support can increase the power consumption twice or more compared with the same test without vectorization. However the performance is improved greatly which, at the end, increases the efficiency.

§6 Related Work

Performance of TBB and OpenMP in scientific and industrial applica-

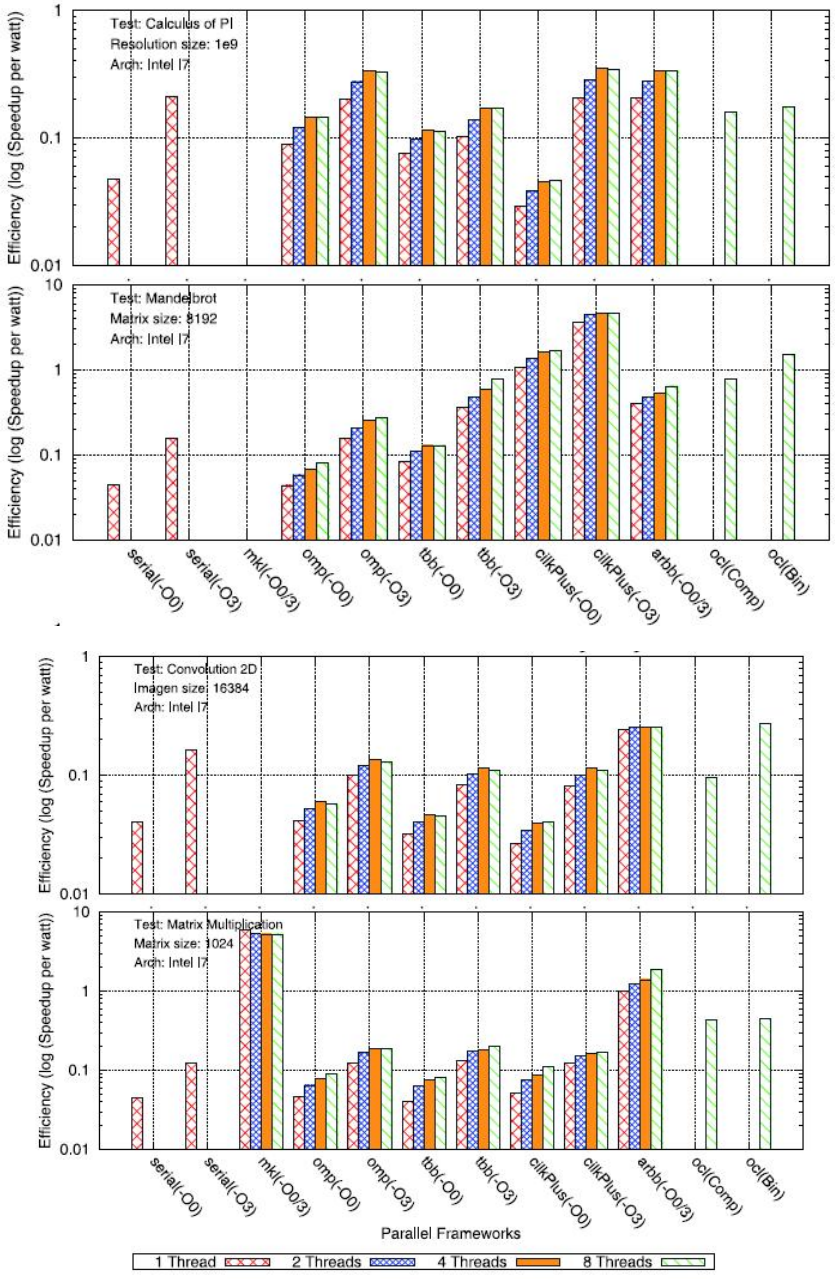

Fig. 4 Efficiency Varying the Number of Threads by using Likwid

Fig. 4 Efficiency varying the number of threads by using Likwid

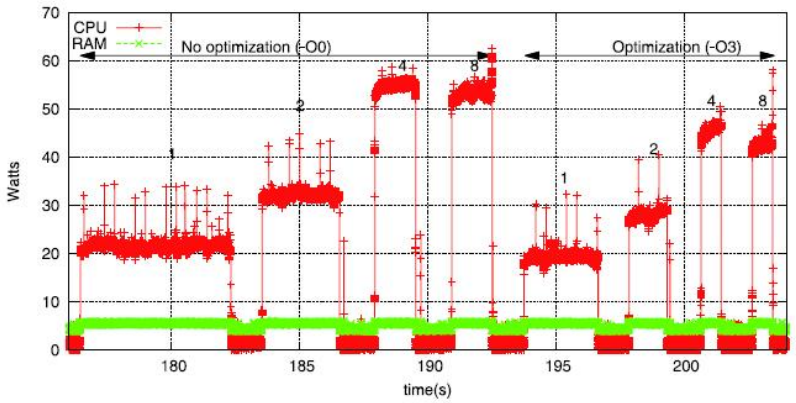

Fig. 5 Power Consumption Varying the Number of Threads for π Benchmark with OMP

Fig. 5 Power consumption varying the number of threads for π benchmark with OMP.

tions has been studied by several authors. In [18], the authors studied several implementations using OpenMP and TBB in a medical processing application, changing the locking mechanisms in the critical sections of the code. As result of these analysis, they concluded that OpenMP is slightly outperforms TBB. However, in [2] the results show that TBB outperforms OpenMP. In this paper, an exhaustive low-level analysis explains that TBB succeeds due to producing efficient code for a substring-finder benchmark. Other authors [15] exploit the behavior of task programming with OpenMP and TBB running on ccNUMA architectures, where TBB uses work-stealing task scheduler [3] to improve data locality. In [1] also compare these technologies with OpenCL [17] including a usability and portability evaluation. They conclude that the latter helps to create highly portable code, at the cost of a greater development effort to achieve optimal performance for each hardware architecture (CPU/GPU).

A correct use of the compiler is essential to build an efficient binary executable, increasing the performance as shown [9] and [6].

The new supercomputers included in top500 [21] make use of these technologies to increase performance. However, When considering power consumption or energy efficiency these architectures are not optimal for parallel computing. Thus the study of new CPUs with new vector instructions, such as AVX2, becomes a interesting issue [22]. For these reasons, the study of new CPU architectures with the new vector instructions as AVX2 becomes in a potential interest point [16] [23]. Recently, new C/C++ extensions have emerged, allowing

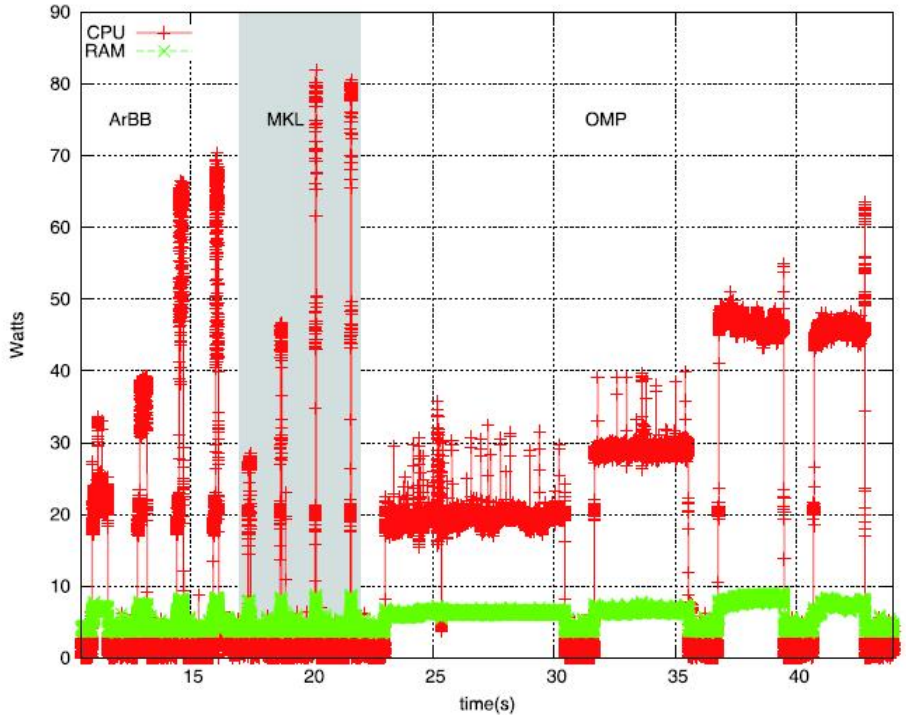

Fig. 6 Power consumption of Matrix Multiplication benchmark using ArBB, MKL and OMP with different number of threads.

for transparent use of SIMD instructions such as [14] or [20].

In addition, an appropriate use of compilers is essential to build an efficient binary executable, thus increasing performance, as shown in [9] and [6].

§7 Conclusions

In this paper we have compared several parallel programming frameworks (OpenMP, TBB, Cilk Plus, ArBB and OpenCL) which are oriented towards shared-memory parallel architectures. We have performed a qualitative comparison based on several factors like task deployment, vectorization support, task scheduling, programming language abstractions and configuration capabilities. Furthermore, we have evaluated they usability, by implementing several parallel benchmarks and measuring the cost of adapting the original sequential code to each one of these frameworks. Finally, we have evaluated their performance executing these benchmarks over different hardware architectures and we have measured how efficient they are analyzing the power consumption on the

execution.

The main conclusion of this study is that each framework have different characteristics that made some more fitted that others depending on the scenarios. If the goal is to perform a quick improvement of a sequential code making it parallel, then OpenMP and Cilk Plus are the correct choice. If we want our parallel code to be object-oriented then TBB is the answer. ArBB is the easiest one to code complex array operations. OpenCL is by far the more complex to use but if the portability between CPU-GPU is required is the only choice.

Furthermore, according to the evaluation, TBB is the best choice when the size of the problem is small due to its low overload. ArBB stands when the problem involves operating with arrays and it is also the best option to execute the same code in different CPUs without recompiling. Also, those problems that involve parallel reduction operations with complex data types are better served using TBB but if the reduction involves simple data types then OpenMP and Cilk Plus are better choices.

Finally, the energy consumption and efficiency results show that CPU vector operations are much more efficiency than multiple parallel threads. Although parallel threads can handle a large number of scenarios where vector operations do not work, like irregular computation problems or irregular data access problems. So, applications that can handle a vectorization approach should employ it to improve their efficiency.

§8 Acknowledgment

This work has been partially funded by the project "Input/Output Scalable Techniques for distributed and high-performance computing environments" of MINISTERIO DE CIENCIA E INNOVACION, TIN2010-16497. The work of J. Daniel Garcia has been funded by "Fundación Cajamadrid" through a grant for Mobility of Madrid Public Universities Professors.

References

1) A. Ali, U. Dastgeer, and C. Kessler, "OpenCL for programming shared memory multicore CPUs," in *In Proceedings of MULTIPROG-2012*, 2012.

2) A. Marowka, "On performance analysis of a multithreaded application parallelized by different programming models using intel vtune," in *Parallel Computing Technologies*, 2011, vol. 6873, pp. 317–331.

3) A. Robison, M. Voss, and A. Kukanov, "Optimization via reflection on work stealing in tbb." in *IPDPS*. IEEE, 2008, pp. 1–8.

4) B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

5) C. J. Newburn, B. So, Z. Liu, M. D. McCool, A. M. Ghuloum, S. D. Toit, Z.-G. Wang, Z. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang, "Intel's array building blocks: A retargetable, dynamic compiler and embedded language," in *CGO*, 2011, pp. 224–235.

6) C. Kim, N. Satish, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey, "Closing the Ninja Performance Gap through Traditional Programming and Compiler Technology," *Intel White Paper 2012*, 2012.

7) G. Contreras and M. Martonosi, "Characterizing and improving the performance of intel threading building blocks," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, sept. 2008, pp. 57 –66.

8) Intel MKL. (2012, Jan.) Intel Math Kernel Library. [Online]. Available: http://software.intel.com/en-us/intel-mkl

9) J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W.-m. W. Hwu, "Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, ser. CGO '10. New York, NY, USA: ACM, 2010, pp. 111–119.

10) J. C. Munson and S. G. Elbaum, "Code churn: A measure for estimating the impact of code change," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 24–.

11) J. Mair, K. Leung, and Z. Huang, "Metrics and task scheduling policies for energy saving in multicore computers," in *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference*, 2010, pp. 266–273.

12) J. Reinders, *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

13) J. Treibig, G. Hager, and G. Wellein, "LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments," in *ICPP Workshops*, 2010, pp. 207–216.

14) M. Pharr and W. R. Mark, "ispc: A SPMD Compiler for High-Performance CPU Programming," in *In Proceedings of Innovative Parallel Computing (InPar)*, 2012.

15) M. Wittmann and G. Hager, "Optimizing ccnuma locality for task-parallel execution under openmp and tbb on multicore-based systems," *CoRR*, vol. abs/1101.0093, 2011.

16) N. G. Dickson, K. Karimi, and F. Hamze, "Importance of explicit vectorization for cpu and gpu software performance," *J. Comput. Physics*, vol. 230, no. 13, pp. 5383–5398, 2011.

17) OpenCL. (2012, Jan.) Open Computing Language. [Online]. Available: http://www.khronos.org/opencl

18) P. Kegel, M. Schellmann, and S. Gorlatch, "Using openmp vs. threading building blocks for medical imaging on multi-cores," in *Euro-Par 2009 Parallel Processing*, 2009, vol. 5704, pp. 654–665.

19) R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," *SIGPLAN Not.*, vol. 30, no. 8, pp. 207–216, Aug. 1995.

20) R. Leißa, S. Hack, and I. Wald, "Extending a C-like language for portable SIMD programming," in *PPOPP*, 2012, pp. 65–74.

21) Top500. (2011, Nov.) Supercomputer sites. [Online]. Available: http://top500.org/lists/2011/11

22) V. Kindratenko and P. Trancoso, "Trends in High-Performance Computing," *Computing in Science and Engineering*, vol. 13, pp. 92–95, 2011.

23) V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 451–460, Jun. 2010.