Future Generation Computer Systems 125 (2021) 743–757

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

The NAS Parallel Benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures

Júnior Löff[a], Dalvan Griebler[a,b,*], Gabriele Mencagli[c], Gabriell Araujo[a], Massimo Torquati[c], Marco Danelutto[c], Luiz Gustavo Fernandes[a]

[a] School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, 90619–900, Brazil
[b] Laboratory of Advanced Research on Cloud Computing (LARCC), Três de Maio Faculty (SETREM), Três de Maio, Brazil
[c] Department of Computer Science, University of Pisa, Pisa, I-56127, Italy

ARTICLE INFO

Article history:
Received 19 March 2021
Received in revised form 2 July 2021
Accepted 14 July 2021
Available online 19 July 2021

Keywords:
NAS Parallel Benchmarks
Parallel programming
Multicore architectures
Performance evaluation

ABSTRACT

The NAS Parallel Benchmarks (NPB), originally implemented mostly in Fortran, is a consolidated suite containing several benchmarks extracted from Computational Fluid Dynamics (CFD) models. The benchmark suite has important characteristics such as intensive memory communications, complex data dependencies, different memory access patterns, and hardware components/sub-systems overload. Parallel programming APIs, libraries, and frameworks that are written in C++ as well as new optimizations and parallel processing techniques can benefit if NPB is made fully available in this programming language. In this paper we present NPB-CPP, a fully C++ translated version of NPB consisting of all the NPB kernels and pseudo-applications developed using OpenMP, Intel TBB, and FastFlow parallel frameworks for multicores. The design of NPB-CPP leverages the Structured Parallel Programming methodology (essentially based on parallel design patterns). We show the structure of each benchmark application in terms of composition of few patterns (notably Map and MapReduce constructs) provided by the selected C++ frameworks. The experimental evaluation shows the accuracy of NPB-CPP with respect to the original NPB source code. Furthermore, we carefully evaluate the parallel performance on three multi-core systems (Intel, IBM Power, and AMD) with different C++ compilers (gcc, icc, and clang) by discussing the performance differences in order to give to the researchers useful insights to choose the best parallel programming framework for a given type of problem.

1. Introduction

Parallel programming enables high-performance applications to leverage the capabilities offered by modern parallel hardware, i.e. shared-memory architectures like multicores and NUMA of multicores, and distributed-memory architectures like clusters, where shared-memory nodes are interconnected via fast networking technologies. The complexity of the available hardware has increased considerably over the years, with processors enhanced with out-of-order computing capabilities, memory hierarchies composed of different coherent private and shared levels of caches, and interconnection networks equipped with smart network interface cards used as co-processors for accelerating networking tasks.

In this ecosystem of complex hardware resources, parallel programming has evolved with the availability of frameworks that support the user in the development of parallel applications and in taking advantage of the underlying hardware in a productive way. Nonetheless, the so-called *programmability wall* [1] is still the main challenge in parallel programming. Developing parallel applications productively requires *high-level programming tools* that hide the complexity of coping with complex hardware. Furthermore, they should enforce *performance portability*, e.g., they shall achieve satisfactory performance on different machines, both in terms of absolute performance like IPS/FLOPS as well as scalability with more processes/threads composing the parallel computation.

For shared-memory architectures, Intel TBB [2] (in C++) and OpenMP [3] (C++ and multi-language support) are popular programming solutions, with the former used for *task-based parallel programming* while the latter to annotate the sequential code with *pragma-based directives* for loop parallelization and, starting from the 4.0 standard, providing a *task-based model* similar to TBB. In addition, the research community has proposed other

[*] Corresponding author at: School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, 90619–900, Brazil.

E-mail addresses: junior.loff@edu.pucrs.br (J. Löff), dalvan.griebler@pucrs.br (D. Griebler), mencagli@di.unipi.it (G. Mencagli), gabriell.araujo@edu.pucrs.br (G. Araujo), torquati@di.unipi.it (M. Torquati), marcod@di.unipi.it (M. Danelutto), luiz.fernandes@pucrs.br (L.G. Fernandes).

https://doi.org/10.1016/j.future.2021.07.021
0167-739X/© 2021 Elsevier B.V. All rights reserved.

families of programming tools fostering high-level abstractions. Notably, algorithmic skeletons [4–6] (generally belonging to the field of Structured Parallel Programming), applied by some C++ research frameworks like FastFlow [7] and SkePU [8], are worthy of consideration, with active communities promoting this parallel programming style.

Assessing the effectiveness of parallel programming frameworks in terms of performance portability on new hardware architectures, which have become available day by day, requires proper *benchmark suites* composed of workloads with sufficiently heterogeneous features to stress different hardware components/sub-systems (e.g., caches, floating-point units, memory bandwidth). Examples of benchmark suites are PARSEC [9], SPLASH [10], SPEC [11], which include workloads from different domains but all focusing on High-Performance Computing.

Among the existing benchmark suites, we are interested in the NAS Parallel Benchmarks [12] (briefly, NPB). NPB is a set of programs originally developed by NASA Advanced Supercomputing division to evaluate the performance of parallel machines. The benchmarks in the suite (three pseudo-applications and five basic kernels) are derived from Computational Fluid Dynamics (CFD) models and can be configured to work with predefined problem sizes (called "classes"). The official implementation of NPB is written in Fortran, with parallel programs developed in OpenMP (based on the Fortran code) for shared-memory systems and MPI (Message Passing Interface) for clusters. NPB has been used over the years for different kinds of research activities. An overview of such works will be given in Section 2.2. Despite the wide use of NPB, no comprehensive porting of all the kernels and pseudo-applications in C++ has been released and made publicly available. The C++ language has become the standard for implementing high-performance code nowadays, and this justifies the effort conducted in this paper to provide a full C++ version of NPB (named NPB-CPP) for evaluating C++ parallel programming frameworks.

The main goal of this paper is to show how the whole NPB suite can be designed and implemented using the structured parallel programming approach, by describing how each NPB kernel and pseudo-application can be designed in terms of composition of the parallel patterns available in modern C++ parallel frameworks. This contribution has a two-fold justification: (i) it allows the evaluation of the expressive power of pattern-based approaches to parallel programming, in analogy with prior works in the literature but based on a benchmark suite never studied before in this perspective (e.g., the work in [13] was based on PARSEC [9]); (ii) the patterned description of NPB allows it to be easily implemented with several different C++ parallel programming frameworks providing the same used patterns, which represents a significant code refactoring effort paying off in terms of code modularity.

The source code of NPB-CPP is provided within a repository made accessible to the community.[1] NPB-CPP is licensed under the terms of the MIT license and has recently been used by other independent research works as a baseline for experimental comparisons [14–20]. It highlights the importance of having a reliable C++ implementation of the distinguished NPB suite. Other works [21,22] are using NPB-CPP for assessing different parallel architectures and programming abstractions, showing NPB's extensiveness for supporting sophisticated evaluations on specialized computer architectures and systems, such as GPUs and clusters.

The main research contributions of our work can be summarized as follows:

- We provide a *thorough* translation (i.e. which respects the original sequential code structure in terms of data structures, loops and original programming style) of the five kernels and the three pseudo-applications from the original serial Fortran code to C++, making NPB-CPP usable and extendable also for future works.
- We implement the entire benchmark suite with different C++ parallel frameworks. In addition to the original OpenMP version (currently maintained by NAS[2]), which is available in Fortran, we provide novel parallel implementations with Intel TBB and FastFlow, aiming at covering both a consolidated and a research-based parallel programming framework.
- We show how the concept of structured parallel programming [23] can be applied for efficiently parallelizing NPB-CPP. We model each one of the parallel benchmarks in NPB-CPP using compositions of Map and MapReduce parallel patterns. This is a higher-level programming approach whose expressiveness and flexibility is discussed in this paper for OpenMP, Intel TBB and FastFlow.
- We provide a careful analysis of the performance obtained by NPB-CPP on different multicore machines and compilers, demonstrating the quality of the porting from the (sequential and parallel) performance and functional correctness perspectives. Experiments were carried out on a set of platforms (Intel Xeon, AMD Epyc, and IBM Power8) to show the performance portability on different multi-core architectures.

It is worth mentioning that our research work has first started in [24]. For this article, significant changes and improvements are made: (i) by re-implementing all the sequential kernels to be compliant with the last NPB version (v3.4); (ii) we complete the sequential porting with the three pseudo-applications that were not studied in our prior work; (iii) we re-implement the five kernels to adhere to the structured parallel programming style, so building the new kernel implementations and the three pseudo-applications parallel code, in terms of parallel patterns (Map and MapReduce), while our prior parallel implementations were based on low-level parallelization approaches; (iv) NPB-CPP can be configured to use all the workload sizes (classes) of the original suite, while the kernels in our prior work supported only small class sizes (up to class B).

The outline of the paper is the following. Section 2 provides a summary of the NPB kernels and pseudo-applications to make the paper self-contained, and provides an overview of past research activities where NPB has been used in the experimental evaluation for different purposes. Section 3 discusses our implementation, both the sequential porting from Fortran to C++ and the design of parallel versions using C++-based parallel frameworks. Section 4 shows and discusses the results of our experimental analysis. Section 5 presents a summary of our findings, while Section 6 concludes our work.

2. Background

In this section, we briefly explain the basic structure of NPB in terms of kernels and pseudo-applications. Then, we will review some recent papers that used NPB.

[1] NPB-CPP's source code: https://github.com/GMAP/NPB-CPP.

[2] https://www.nas.nasa.gov/publications/npb.html.

2.1. NAS Parallel Benchmarks (NPB)

The NPB suite has five kernels and three simulated applications. The code poses several different challenges, as far as performance is concerned, such as irregular memory accesses, complex data dependencies, and short- and long-distance data communications [12], where the former stresses data locality while the latter memory bandwidth capacity. The five kernels, all in Fortran except IS which is in C, are described as follows:

- **Embarrassingly Parallel (EP)**. It generates a large number of Gaussian random deviates and enumerates them to compute the Gaussian deviation by utilizing the acceptance–rejection method. Finally, the number of pairs that lie in the square annulus is computed. This method is useful to measure the capacity of the floating-point operations of the target architecture [12].
- **Multi Grid (MG)**. It utilizes the V-cycle MultiGrid method to compute a 3D scalar Poisson equation where the kernel continuously computes restriction and prolongation when alternating between coarse and fine grids. The goal is to stress short- and long-distance data communications [12, 25].
- **Conjugate Gradient (CG)**. It computes an approximation of the smallest eigenvalue of a large, sparse, and unstructured matrix, utilizing the Conjugate Gradient method. This kernel stresses data communication mechanisms as well as memory locality and caches [25].
- **Discrete 3D Fast Fourier Transform (FT)**. It computes a Fast Fourier Transform (FFT) of a 3D partial differential equation using the spectra and inverse methods in an iterative loop. It simulates an intensive long-distance communication [12, 25].
- **Integer Sort (IS)**. It performs an integer sorting among a sparse set of numbers, which simulates an important computation for particle-in-cell applications. By default, it is based on the Bucket-Sorting algorithm. This kernel simulates and measures integer computation and data communication capabilities [12].

The pseudo-applications implement three different iterative methods to solve a 3D Navier–Stokes system of differential equations describing the flow of incompressible fluids. We summarize them as follows:

- **Block Tri-diagonal solver (BT)**. It is an expensive implicit algorithm to numerically solve 3D Navier–Stokes equations. The solution is based on an Alternating Direction Implicit (ADI) factorization on a 3D matrix, which produces block-tridiagonal systems that, along each direction, solve the unknown vectors using the back substitution method [25].
- **Scalar Penta-diagonal solver (SP)**. It uses the Beam-Warming approximate factorization to decompose the 3D matrix. The output consists of a particular case of band matrices known as Scalar Pentadiagonal matrices. Then, the tridiagonal matrix algorithm is applied over the block-tridiagonal systems and the back substitution method solves the remaining vectors [25].
- **Lower-Upper Gauss–Seidel solver (LU)**. It utilizes the Symmetric Successive Over-Relaxation (SSOR) method, which combines two SOR computations. The latter is a variation of the Gauss–Seidel method that solves a linear system of equations. First, a forward SOR sweep is performed followed by a backward SOR sweep to update the unknown variables in reverse order.

2.2. NPB in prior works

The current available official NPB version is implemented in Fortran, which was elected over other languages considering its popularity in CFD applications. The selected papers for discussing and comparing are those based on C language. Previous works, such as [26,27] and [28], have used NPB for evaluating specific architectures and systems. The authors of [26] have ported the NPB to C and re-implemented the applications in OpenCL, to leverage heterogeneous architectures equipped with GPUs. However, no information about how the conversion was performed is given. Also, the authors did not consider the largest NPB workloads (e.g., class C) because of the limited memory capacity of the GPU considered (GeForce GTX 480). More recently, a new research has extended this work by proposing an improved GPU version for OpenCL and CUDA [27]. Again, no information about the pitfalls of the porting was provided, while the focus was only on the parallelization. The performed evaluation demonstrates increased performance compared to the original version on GPU-based platforms, while no results have been presented on standard multicore platforms.

A prior work [28] presents a methodology to improve a strong scalability evaluation for OpenMP. They implemented the methodology in the PCERE (Parallel Codelet Extractor and REplayer) tool that extracts and executes OpenMP parallel regions. Their evaluation was performed on a C version developed starting from an early NPB version (2.3) [29], which was later made available. Some research works have focused on the Unified Parallel C (UPC) language, an extension of the C programming language designed for HPC platforms. A version of NPB was ported to the UPC language, and it is composed of two main parts. The first contains all the five NPB kernels and was developed by the HPC Laboratory from the George Washington University as part of the Berkeley UPC Compiler project [30,31]. The second contains the three pseudo-applications and was developed by the NASA Ames Research Center [32]. This NPB-UPC version is distributed with the Berkeley UPC project. The five previous research works [26–28,30,32] put a significant effort to convert and extend the NPB towards C-based languages.

In recent studies, C-based NPB versions have been used for evaluation purposes. The work in [19] investigates the performance and potential of the parallel STL (Standard Template Library) using NPB kernels. Parallel STL is an Intel library developed using Intel TBB as a backend for parallel algorithms. This work used our previous version of the NPB ported to C++ [24], which was limited to the five kernels without the pseudo-applications. It also highlights the importance of having such sequential porting in C++. In our work, besides completing the NPB porting, we study the performance of NPB directly implemented using Intel TBB without the additional layer provided by Parallel STL, for a fairer comparison with OpenMP and FastFlow.

The work in [20] uses the NPB to evaluate five auto-parallelizing compilers (Cetus, Par4all, Rose, ICC, and Pluto). Also in this case, the authors used our previous version of the NPB ported to C++. An optimized memory allocator for the Single-Assignment C (SaC) compiler was proposed in [33]. The authors re-implemented the NPB in SaC to evaluate their optimizations. A tool that systematically analyzes shared-memory accesses of UPC applications has been proposed in [34]. The authors tested NPB kernels and applications to fine-tune data redistribution, and for enhancing the use of private variables for improving local accesses. A locality-aware framework for thread affinity placement based on hierarchical data locality has been presented in [35]. The authors selected four NPB kernels (IS, FT, CG, and MG) for covering a wide range of communication patterns. Hardware support mechanisms to efficiently manipulate PGAS address mapping and to improve data access overhead have been introduced in [36]. Experiments were conducted on a subset of NPB kernels.

Among the previously cited papers, we selected five papers that are closely related to our work for comparing specific features. Table 1 provides a summary of the characteristics of those five papers and this work. The first and second columns report the reference and the year of publication. The third column reports the target kernels and pseudo-applications supported. The fourth column shows the NPB version on which they are based. The fifth column shows the language used and the sixth column reports the adopted parallel programming frameworks. The seventh column highlights the targeted parallel architectures.

In past papers [26,27], the authors have translated all the kernels and applications of the NPB 3.3 targeting the C language. In [28] and [30] authors used an outdated NPB version. In [32] authors have implemented three pseudo-applications using the NPB 3.3 version to target the UPC programming language. Differences are also observed concerning the target architectures. Although [28] is targeting multicores as our work, their version is a raw translation to test a compiler tool. Most significantly, none of these previous works aimed to provide a consistent and generic NPB version (kernels and pseudo-applications) in C++ that is platform agnostic (see Section 3). Our benchmark code can be easily extended to support other computer architectures through the use of existing C++ parallel programming frameworks. Furthermore, new C++ compiler tools can also use our NPB-CPP to evaluate the impact of new code optimization and auto-parallelization techniques, which represents an additional, indirect contribution of our work.

2.3. Parallel programming frameworks

Several consolidated parallel programming frameworks are available with the C++ programming language. Most of them are based on the structured parallel programming paradigm, in which few patterns/constructs are provided as basic building blocks to develop easily and productively parallel code (e.g., parallel-for, reduce, map, and so forth). In the following, we introduce three popular parallel programming frameworks: OpenMP, Intel TBB and FastFlow.

OpenMP. OpenMP [3] is a parallel programming framework for multicores. It is based on *pragma* annotations to be applied directly on the source code before regions that take a significant portion of the overall execution time, like *for* and *while* loops. The OpenMP compiler is in charge of transforming the annotations in a code leveraging multicores by using threads that split the execution of the loop iterations. Recently, starting from the `4.0` standard, OpenMP supports the *task-based parallelism paradigm* with proper pragmas to create tasks and link them with dependencies. This allows the programmer to create a task, e.g. a portion of code plus its surrounding data environment, which can be scheduled for the execution on an available thread provided that all its input dependencies are satisfied.

Intel TBB. TBB [2] is the Intel suite for parallel programming on multicores. In the original idea, TBB provides the programmer with abstractions to create tasks connected by dependencies and to schedule them transparently on a pool of threads. Complex work-stealing techniques have been adopted to balance the parallel workload in an effective and cache-friendly manner. A higher level of abstraction has been added to represent graphs of special importance in the parallel programming practice, like pipelines of filter stages.

FastFlow. FastFlow [7] is an open source structured parallel programming framework. It provides the application programmer with a variety of ready-to-use stream and data parallel patterns that may be freely composed and customized to implement complex parallel applications. FastFlow is a header-only library implemented on top of POSIX Pthread and C++11 features, and parallel patterns are used by instantiating proper classes of the library. The framework has been designed to target multicores with two main goals in mind: *performance* and *programmability*.

3. Implementation

The goal of this section is twofold. First, we describe how the porting has been developed, and the principles behind our thorough translation from Fortran (C in case of IS) to C++. Then, in the second part, we show the strategies we followed for parallelizing NPB with different C++ parallel programming frameworks. Our design choices aim at simplifying the usability and extensibility of NPB-CPP. Therefore, we provide C++ code that can be easily converted to C-like code with minimal modification effort. We target C++ because there is a plethora of important solutions available only for this programming language. Many of these solutions [37] exist for a long time and yet never had the opportunity to be evaluated using the NPB suite.

3.1. C++ porting conventions

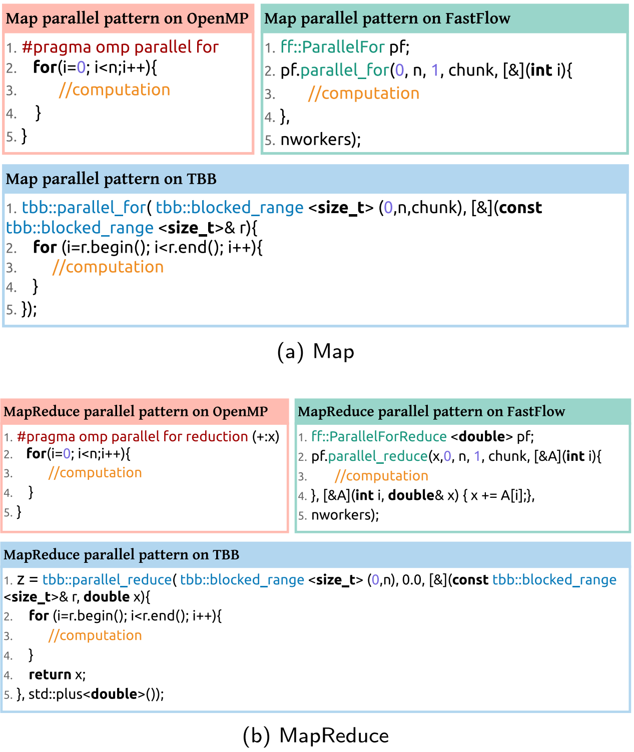
Our porting was conducted following the official documentation [12,25]. When implementing the C++ code of NPB, the first guideline was: every time a global array is declared in the Fortran code, we allocate it as dynamic memory in C++. Static memory accesses are faster than dynamic memory accesses [38]. However, dynamic memory allocation is a recommendation from the official reports of NPB since it uses very large arrays that may result in memory stack overflow errors. Also, dynamic memory allocation is the main feature introduced in the NPB 3.4 version. Furthermore, we implemented the global arrays to be allocated in one single dimension for all kernels and pseudo-applications. However, for internal multi-dimensional accesses, we perform conversions from linear to multidimensional indexes. NPB reports do not recommend the use of fixed multidimensional arrays because the benchmarks implement different data access patterns. In our design, we use linear arrays as the main layout for our data structures because, although it is not necessarily the best choice to optimize the cache hierarchy utilization (given the irregular access pattern of some of the benchmarks), our goal is to provide a generic code that can be extensible for a wide spectrum architectures. Nonetheless, the user can disable those features through a compiler flag, where static multidimensional arrays are created.

Another important guideline was to follow as much as possible the original Fortran code semantics during the sequential code porting. However, it was not always possible to literally translate the code. In particular, we often needed to modify the ordering of accesses to arrays. The reason is that Fortran is column-major ordered (each complete column of the matrix is stored before the next one), while C++ is row-major ordered. NPB code has also many `goto` statements, which have been removed by implementing `while` and `for` loops with proper stop conditions. Such change is useful for different reasons: first, because the use of `goto` is a legacy feature in C++ (inherited from raw C); second, the use of traditional loops helps the introduction of parallel patterns (e.g., parallel-for and map) during the parallelization described later in the paper.

Although the IS kernel was already implemented in C, it uses some Fortran routines shared with other kernels. During our porting to C++, we re-implemented such routines in C++. The porting

Table 1
Summary of past papers using NPB in their experimental evaluation.

Reference	Year	Benchmarks	NPB version	Languages	Parallel versions	Target architecture
[30]	2002	CG, EP, FT, IS, MG	NPB 2.4	UPC	UPC	Cluster
[32]	2009	BT, LU, SP	NPB 3.3	UPC	UPC	Cluster
[26]	2011	BT, CG, EP, FT, IS, LU, MG, SP	NPB 3.3	C	OpenMP, OpenCL	GPU
[28]	2015	BT, CG, EP, FT, IS, LU, MG, SP	NPB 2.4	C	OpenMP	Multicore
[27]	2019	BT, CG, EP, FT, IS, LU, MG, SP	NPB 3.3	C	OpenMP, OpenCL, CUDA	GPU
Ours	2021	BT, CG, EP, FT, IS, LU, MG, SP	NPB 3.4	C++	OpenMP, FastFlow, TBB	Multicore



Map parallel pattern on OpenMP

```
#pragma omp parallel for
  for(i=0; i<n;i++){
      //computation
  }
}
```

Map parallel pattern on FastFlow

```
ff::ParallelFor pf;
pf.parallel_for(0, n, 1, chunk, [&](int i){
      //computation
},
nworkers);
```

Map parallel pattern on TBB

```
tbb::parallel_for( tbb::blocked_range <size_t> (0,n,chunk), [&](const
tbb::blocked_range <size_t>& r){
  for (i=r.begin(); i<r.end(); i++){
      //computation
  }
});
```

(a) Map

MapReduce parallel pattern on OpenMP

```
#pragma omp parallel for reduction (+:x)
  for(i=0; i<n;i++){
      //computation
  }
}
```

MapReduce parallel pattern on FastFlow

```
ff::ParallelForReduce <double> pf;
pf.parallel_reduce(x,0, n, 1, chunk, [&A](int i){
      //computation
}, [&A](int i, double& x) { x += A[i];},
nworkers);
```

MapReduce parallel pattern on TBB

```
z = tbb::parallel_reduce( tbb::blocked_range <size_t> (0,n), 0.0, [&](const tbb::blocked_range
<size_t>& r, double x){
  for (i=r.begin(); i<r.end(); i++){
      //computation
  }
  return x;
}, std::plus<double>());
```

(b) MapReduce

Fig. 1. Basic routines to implement the Map and MapReduce parallel patterns using OpenMP, FastFlow, and Intel TBB.

to the FT kernel required a substantial code refactoring effort to remove the batch mechanism which gathers a set of FFT operations to increase the computational granularity. Consequently, there is no need to manually specify the batch size parameter. Other code modifications (e.g., removing data dependencies, nested loops, low-level cache optimizations) were necessary to keep the coherence between the sequential and OpenMP code structures for FT and IS since they were the only kernels having different sequential and OpenMP versions. This contributes to have a fair performance comparison when parallelizing the code. Finally, this set of features provided in our porting facilitates the implementation of the NPB in other architectures such as GPUs [21], lowering the amount of refactoring. For instance, GPU's parallel software development require linear arrays that are already provided in our NPB version, otherwise, the programmer would need to provide additional routines for transforming the data layout. Those features are not provided by other works in the literature [26,28].

3.2. Parallel implementation

The original NPB code in Fortran is shipped with an official OpenMP parallel version optimized by NAS experts. The NPB-CPP's OpenMP parallel version follows a structured parallel programming approach with Map and MapReduce data-parallel patterns. Our C++ porting allows the evaluation of implementations based on different parallel programming frameworks available in C++. As said before, we consider Intel TBB [2] and FastFlow [7]. We chose FastFlow because we compare it against state-of-the-art solutions for shared-memory architectures like OpenMP and TBB, and the results of this comparison also help to improve other higher-level APIs and DSLs (Domain-Specific Languages) that rely on FastFlow as its parallel runtime [39–41].

3.2.1. Parallel patterns in NPB-CPP

Parallel patterns can be separated in two concepts: their high-level semantics and their low-level implementation strategies. The semantics concern *when* and *where* a parallel pattern can be used in the sequential code and its possible usage limitations (e.g., to avoid breaking the sequential code semantics). The pattern implementation instead is rich of low-level details abstracted to the user, such as which kinds of communication queues, synchronization protocols, and scheduling decisions are used internally to implement the pattern. By looking at the high-level semantics, a programmer is able to parallelize an application without getting involved with low-level and complex parallelism details. Each parallel programming framework offers its own implementation of parallel patterns (and a slightly different API to instantiate them). Indeed, parallel patterns can vary in terms of usability and efficiency since they may use contrasting interface design, and internal implementations.

We will adopt the following guidelines during the parallelization design: (1) the first goal is to follow the structured parallel programming approach and parallelize the benchmark applications using only the Map and MapReduce data-parallel patterns (Fig. 1) (2) our second goal is to be uniform in the design, by providing semantically equivalent implementations as well as making use of the features available. For example, although we have intrinsic mechanisms such as `std::mutex` in C++, we choose `tbb::mutex` from Intel TBB; (3) our third goal is to avoid architecture-specific optimizations and let the code be portable on a range of different platforms. However, people interested to obtain the maximum performance on specific architectures or parallel programming frameworks may in the future easily extend NPB-CPP to implement their specific optimizations. Examples of them are the use of custom task-parallelism patterns, memory and thread affinity strategies, or even targeting different platforms like distributed architectures (e.g., clusters) or GPUs (see [21]).

OpenMP, TBB, and FastFlow provide different parallel programming APIs to instantiate their own patterns. In order to generalize the presentation, we describe them using abstract *Map* and *MapReduce* data-parallel patterns as well as *Critical Sections* and *Barriers* that are all present in the different frameworks with different internal implementations. They are introduced in the following list:

- The **Map** pattern [23] consists of the replication of a function that applies over all elements of an indexed set. This can be used to parallelize `for` loops when iterations are independent. OpenMP, TBB, and FastFlow offer an API called "parallel for" for this purpose, see Fig. 1(a). In OpenMP, programmers annotate parallelizable `for` loops using compiler pre-processor directives. In TBB and FastFlow, programmers replace parallelizable `for` loops using routines that are implemented by a C++ template library. The main difference is that the parallel region for OpenMP and TBB has the thread-private feature while FastFlow does not have. The scheduling type is optional, however, OpenMP and FastFlow apply by default a static assignment of iterations to the underlying threads, while TBB uses a dynamic distribution.

- The **MapReduce** is the union of a Map and a Reduce pattern. According to [23], the Reduce pattern combines all the elements from a collection and produces a single element using an associative binary operator. Therefore, in the MapReduce, every element of the Map is combined into a single element. This pattern can be used to parallelize `for` loops when iterations exhibit specific data dependencies and a synchronization is required. OpenMP, TBB and FastFlow offer an API called "`parallel_reduce`" as shown in Fig. 1(b). In OpenMP, programmers use `reduction` along with the `parallel for` directive for specifying the operation type and the reduced variable. This parameter only accepts predefined types. In TBB and FastFlow, programmers replace the target `for` loop with a specific routine receiving as arguments a set of parameters such as the lambda functions implementing the Map and the Reduce steps.
- The **Barrier** is a synchronization primitive for a group of threads [23]. The barrier guarantees a synchronization point where any thread must stop and cannot proceed until all other threads reach the barrier. The barrier definition considers both implicit and explicit barriers. By default, implicit *barriers* are found at the end of Map and MapReduce patterns on all the considered parallel programming frameworks. OpenMP has a `nowait` directive that allows threads to avoid this synchronization. However, TBB and FastFlow do not have this option. Explicit *barriers* are implemented when synchronization is required across threads. In OpenMP we used the `#pragma omp barrier`. In TBB and FastFlow, we used the standard Pthread library barrier mechanisms since they do not support such low-level programming techniques while it was necessary for expressing parallelism.

Table 2 shows the number of instances of the parallel patterns and synchronization primitives that have been used in our NPB-CPP parallelization with the three parallel programming frameworks. The FastFlow and TBB versions show a different number of patterns and synchronization primitives in some kernels and pseudo-applications (MG, CG, IS, and LU). The reason is that some instances of *Map* and *MapReduce* do not bring any performance improvement (their loop body does very small computation). Better performance can be achieved with the TBB and FastFlow version by removing such fine-grained Map and MapReduce instances. OpenMP presents less overhead and is able to exploit such fine-grained parallelism, mostly paying off in bigger workload classes. Other minor asymmetries in the implementation with the different frameworks are very specific to each kernel and pseudo-application, and we omit to describe them since they do not represent a central point for this discussion.

In the rest of this section, we describe the main aspects of the parallelization for each kernel and pseudo-application.

3.2.2. EP kernel

The EP kernel has a single compute-intensive code region that we parallelized using a *MapReduce* with static scheduling. This choice has been applied in all our implementations for the different parallel programming frameworks. Additionally, this kernel needs a special synchronization at the end of the parallel computation. As the default *MapReduce* implementation (described in Section 3.2.1) accepts only standard types and there is a reduction over an array, we manually implemented the data synchronization in OpenMP, TBB, and FastFlow.

Table 2
Structure of each NPB benchmark in terms of parallel patterns in FastFlow(FF), OpenMP (OMP), and Intel TBB (TBB).

Benchmark	Map			MapReduce			Barriers		
	FF	OMP	TBB	FF	OMP	TBB	FF	OMP	TBB
EP	1	1	1	1	1	1	1	1	1
MG	11	11	10	1	1	1	11	11	10
CG	7	18	7	4	6	4	7	11	7
FT	8	8	8	1	1	1	8	10	8
IS	6	7	6	1	1	1	6	7	6
BT	23	23	23	–	–	–	23	23	23
SP	19	19	19	–	–	–	19	19	19
LU	22	23	22	1	1	1	22	19	22

3.2.3. MG kernel

The MG kernel uses the multigrid V-cycle operation with a residual computation. The strategy adopted is to parallelize with *Maps* the intensive computational regions of the V-cycle method, which are the restriction, prolongation, residual, and smoother routines. Then, we also parallelized some less intensive routines such as the communications along borders (with a *Map*) and the approximation to the L2 and uniform norm values (with a *MapReduce*). This last *MapReduce* needed special care since threads synchronize on two variables for different operation types (sum and max). In OpenMP, this is done with two different `reduction` directives, while in FastFlow and TBB we manually implement the *MapReduce*. Finally, we point out that the MG kernel exhibits limited scalability with more threads, as it will be shown in the experimental part. The reason is that the main computational step works on a small grid and requires non-local accesses to the memory, leading to poor cache exploitation.

3.2.4. CG kernel

The most computationally demanding step in CG is the sparse matrix–vector multiplication $q = Ap$ of the Conjugate Gradient method, which we parallelize using a *Map*. In this case, the static scheduling works well although CG has an irregular workload. This is because the workload follows a random Gaussian distribution, and when the slices are statically divided they tend to store equally balanced workload. Furthermore, we used multiple *Maps* and *MapReduces* to parallelize other less compute-intensive steps, as in Table 2. The consequence of including several *Map* and *MapReduce* is that synchronization barriers are implicitly added to the code, and this synchronization overhead dominates the small improvement obtained by introducing such parallel steps. We mitigated this in OpenMP, where some implicit barriers in the *Maps* were removed using the `nowait` directive.

3.2.5. FT kernel

This kernel contains three independent symmetric FFT routines to compute each dimension, which we parallelized using a *Map*. However, communications may represent a bottleneck since the kernel must decompose slices of the main 3D matrix into a 1D local array each time it applies an FFT resolution, and then it requires to copy the results back. In addition, we manually implemented the *MapReduce* for computing the checksum in all the parallel programming frameworks. This needs special treatments because the reduce operation is applied over complex numbers.

3.2.6. IS kernel

The IS kernel uses the bucket sorting approach. The sequential code (originally written in C) has a synchronization protocol for the parallel OpenMP version. This protocol needs to be adapted for TBB and FastFlow. To this end, we adapted the semantics of the default *Map* (described in Section 3.2.1) by using it just to replicate the same function without specifying which elements of the indexed set are computed in parallel. Then, the scheduling is performed manually inside the threads' private region. This splits the computation and determines which indexes of the set each thread computes. An explicit *Barrier* is added after each instance of the modified *Map* for synchronization. We point out that in Table 2 the number of *Maps* in IS matches the number of *Barriers* for this reason. We also instantiated the standard *Map* in some loops, as for the sorting operation inside the buckets. In this case, we used the dynamic scheduling approach to better balance the workload between buckets. Finally, we instantiated the *MapReduce* to implement the verification function after the computation.

3.2.7. BT and SP pseudo-applications

Implicit methods are typically used to get an approximation of the solution in CFD equations through iterative techniques, as implemented in the BT, SP, and LU pseudo-applications. Besides the divergent factorization methods used for BT and SP, those pseudo-applications were very similarly designed. Both are originally implemented from a parallel viewpoint. Therefore, their data have already been organized to avoid dependencies. In OpenMP, FastFlow and TBB, we only instantiated sequence of *Maps* to parallelize them, as shown in Table 2. A practical example of the data organization's importance concerning performance is found in BT and SP for the residual computation routine. Dimensions x and y are implemented with a single *Map* due to the lack of data dependencies since data are organized along the z direction. However, the z dimension itself is fragmented to avoid data dependencies, requiring six *Maps* instead of a single one as in x and y directions. Finally, the most compute-intensive steps of both BT and SP, along with the residual computation, are the three solving functions for computing each of the three dimensions. We parallelized them instantiating the *Map* pattern in the outermost loops.

3.2.8. LU pseudo-application

The LU pseudo-application implements the Symmetric Successive Over-Relaxation (SSOR) method. This algorithm extends the Gauss–Seidel method to solve a linear system of equations. Its intensive computation relies on the decomposition of the 3D matrix system in triangular lower/upper matrices and then solving this matrix system. Both computations, upper and lower, are performed in two similar steps on the same iteration. Previous studies [25,32] suggest two main ways to parallelize the LU application: *hyperplane* and *pipelining*. In the first, points from the same hyperplane defined by $l = i + j + k$ can be computed in parallel. The so-called *pipeline* strategy instead implements a synchronization structure using data-parallelism mechanisms to control the computational flow in a way that mimics a multi-dimensional *pipeline*. The LU cannot be parallelized efficiently with traditional data-parallelism techniques because all three dimensions must deal with data dependencies. This means that any modification would send a lot of update messages between threads, impairing parallel scalability.

We adapted the original OpenMP parallelization in FastFlow and TBB. First, we modify the *Map* as we did for the IS kernel (see Section 3.2.6). We also implemented synchronization mechanisms using *locks* to control the data flow in FastFlow and TBB, where a thread starts the computation only when its previous neighbor has finished. OpenMP uses a different approach based on `omp flush` directives. We graphically show the parallel data flow outcome in Fig. 2. Parallelism is achieved when for each advance along the k direction, a new thread starts computing the next block of elements from the indexed set.

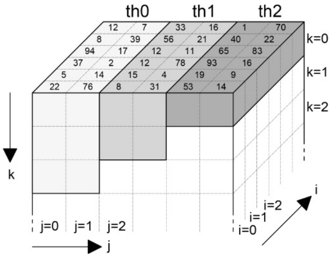


Fig. 2. The parallel data flow illustration for LU.

4. Experiments

The set of experiments was carried out on three different multicore platforms, namely: `Xeon`, `Epyc`, and `Power8`. We used three compilers in the analysis: GNU compiler (`gcc`), Intel compiler (`icc`) and `clang`. The architectural specifications and environment settings of the three machines are reported in Table 3. The NPB workload sizes are expressed through classes, where classes A, B, and C are standard test problems having about 4× larger size increase going from one class to the next one. In the evaluation, we mainly focus on the performance obtained with class C, which represents a good compromise between memory occupancy and computational granularity for our machines. More details about the several parameters that each class problem has can be found in [42]. We also report experimental results with a smaller size (class B) to observe the impact of the runtime overhead characterizing the different C++ parallel programming frameworks. For compiling the benchmarks, we specified `-std=c++14` and `-O3`. Each experiment configuration was repeated 5 times. The plots are reporting the arithmetic mean value and the standard deviation using error bars. We also apply statistical analysis using 95% of reliability to compare the differences in the execution times. We ensure the functional correctness of the results provided by NPB-CPP using the built-in verification functions implemented in the NPB. Such functions compare the results obtained in one execution with the correct ones stored within the benchmark. The execution is considered successful when the result is within a certain error range (e.g., CG tolerates less than 10^{-10}). In all cases, our C++ porting successfully passes all those verification checks.

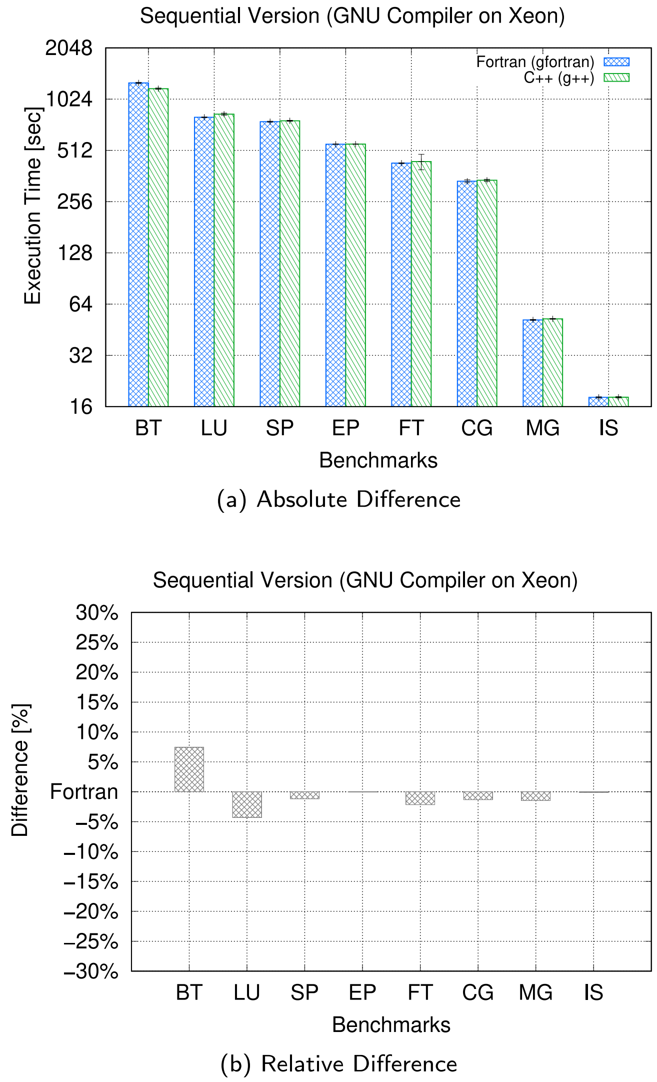
We will first discuss the performance of the sequential versions. Second, we examine the performance of the parallel porting using the same OpenMP runtime (distributed within compilers). Then, we discuss the performance obtained by using Intel TBB (2020.1) and the FastFlow (3.0.0) library. Finally, we compare the performance against a previous implementation of the five kernels developed in our prior work.

4.1. Sequential porting

In this section, the goal is to evaluate our sequential porting, observing the performance behavior of NPB-CPP compared to the original NPB. Although we only present the plots for the `Xeon` platform, we also describe in a the outcome for the other two platforms. Fig. 3 shows the results obtained with the GNU `gfortran/g++` compilers on the `Xeon` platform. X-axis presents the NPB benchmarks in both graphs, while the Y-axis presents

Table 3
Multicore platforms and their environment settings (OS and compilers).

Name	Description
Xeon	It has a Dual-socket Intel Xeon E5-2695 Ivy Bridge CPUs running at 2.40 GHz, featuring 24 cores (12 per socket). Each hyper-threaded core has 32 KB private L1, 256 KB private L2 and 30 MB of L3 shared with the cores on the same socket. The machine has 64 GB of RAM. **System.** Linux `4.15.0-72`, Ubuntu `18.04.3LTS`. **Compilers.** GNU `gcc 9.1.0`, Intel `icc 19.0.5.281`, and `clang 10.0.0`
Epyc	It has a Dual-socket AMD EPYC 7551 CPUs Zen micro-architecture running at 2.40 GHz, featuring 64 cores (32 per socket). Each core has 2 HW threads, 64 KB private L1, 512 KB private L2 and 8 MB of L3 shared with other three cores. Each socket has 4 NUMA nodes. The machine has 128 GB of RAM. **System.** Linux `4.15.0-101`, Ubuntu `18.04.4LTS`. **Compiler.** GNU `gcc 9.1.0`
Power8	It has a Dual-socket IBM server 8247-42L with two Power8 processors each with ten cores organized in two CMPs of 5 cores working at 3.69 GHz. Each core (8-way SMT) has private L1d and L2 caches of 64 KB and 512 KB, and a shared on-chip L3 cache of 8 MB per core. The total number of cores per CPU is 20 physical and 80 logical ones. The machine has 64 GB of RAM. **System.** Linux `4.4.0-47`, Ubuntu `16.04`. **Compiler.** GNU `gcc` version `9.1.0`.



(a) Absolute Difference

(b) Relative Difference

Fig. 3. Comparing the performance of NPB vs NPB-CPP on the `Xeon` platform using the GNU compiler with Class C.

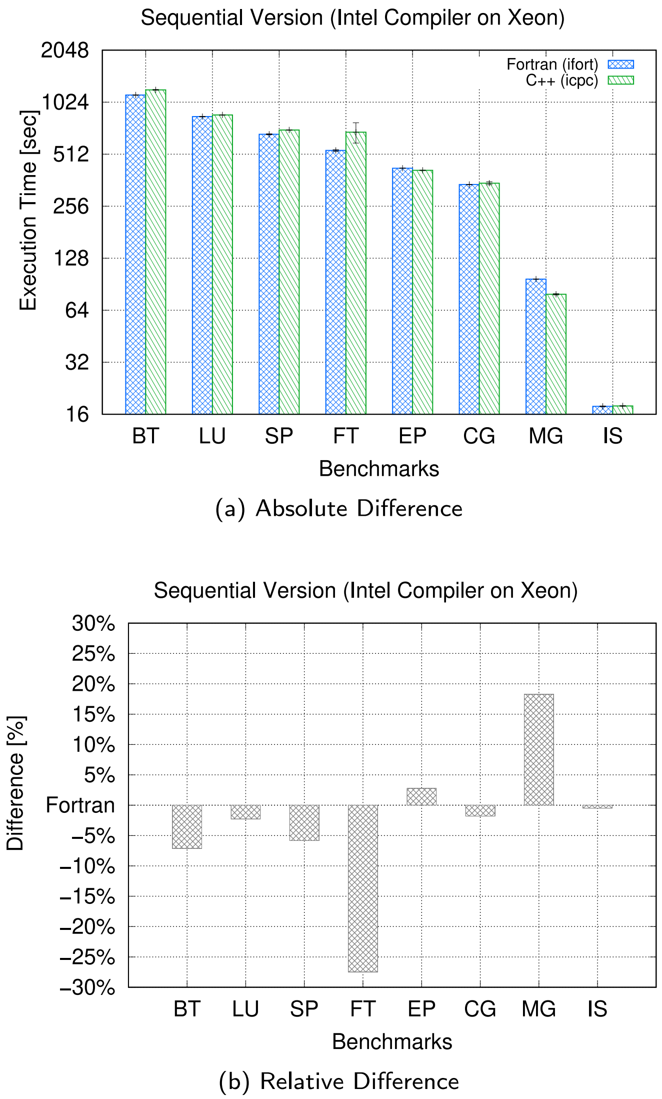
the execution time in seconds (using a logarithmic scale) in one plot while the other shows the relative difference in percentage. The relative difference is obtained by normalizing the NPB-CPP execution times with respect to the original NPB ones, where positive bars stand for C++ faster and negative bars the opposite.

On average, the C++ version achieves similar performance compared with the original Fortran code. The normalized difference is less than 1.5%, except for BT, LU, and FT. One of the major differences between NPB-CPP code with respect to NPB is that we dynamically allocate single dimensional arrays, while in NPB they were allocated statically using multiple dimensions. As already explained in Section 3, we use linear arrays to provide a generic code that can be extensible for a wide spectrum architectures. Also, since the benchmarks implement irregular data access patterns, using linear memory allocation is less intrusive than multi-dimensional arrays for this purpose. In BT, this modification has a positive impact – C++ is 7.43% faster than Fortran – while in LU the impact is negative, since C++ is 4.3% slower. Concerning FT, the C++ version is 2.15% slower than the Fortran version, mainly because of the complex type (intrinsic data type only in Fortran).

To observe the performance using different compilers on the same platform, Fig. 4 shows the execution time in seconds (using a logarithmic scale) obtained with the Intel compiler on the `Xeon` platform. The relative differences between Fortran and C++ are also shown. In this case, the differences between the two versions are more evident. The FT kernel presents the biggest difference: the C++ version is 27.49% slower than the Fortran one. The MG kernel also shows a high normalized difference: the C++ version is 18.26% faster than the Fortran version. Both FT and MG are the two benchmarks that allocate more memory and stress long-distance data communications (see Section 2.1). We observed that the most significant performance differences are in those benchmarks that are more memory intensive. Besides the FT and MG kernels, the C++ version of the BT and SP pseudo-applications are 7.13% and 5.83% slower than the Fortran ones. These pseudo-applications are faster when using `ifort` than `gfortran`, while the C++ version (`g++` and `icpc`) is similar for both compilers, which explains the small increase in the difference. For the remaining benchmarks, the average difference is less than 3%.

The analysis can be extended by considering the impact of a different open-source compiler: `clang`. `clang` does not offer an official Fortran compiler. For this reason, we show in Fig. 5 the performance difference obtained by running NPB-CPP (with class C) using all the available C++ compiler technologies on the `Xeon` platform. As shown in the graph, the performance varies among the compilers and each one is able to optimize the code better than the others depending on the benchmark. Similar to the behavior observed in previous Fig. 4, the biggest variations occur in FT and MG, where both execute memory intensive computations. In FT, the standard deviation is high and `gcc` achieves the best results, where `icc` and `clang` are respectively 55.9% and 31.5% slower than `gcc`. Similarly, in MG the `icc` and `clang` compilers are respectively 50.9% and 19.2% slower than `gcc`. However, in the EP benchmark `gcc` is the fastest and achieves up to 34.8% better execution time than others. In LU, `clang` is up to 24.7% slower than others. In the remaining benchmarks the difference is smaller, achieving less than 10% difference between compilers.

The experiments conducted for the other two platforms using the GNU compiler are summarized as follows. We present the percentage numbers normalizing the difference between NPB-CPP and NPB execution times. The outcome is that NPB's benchmarks are on average 0.86% faster than NPB-CPP ones in `Epyc`. In



(a) Absolute Difference

(b) Relative Difference

Fig. 4. Comparing the performance of NPB vs NPB-CPP on the Xeon platform using the Intel compiler with Class C.

Power8, NPB-CPP's benchmarks are on average 0.73% faster than NPB ones. As discussed for the Xeon platform, the differences are less than 1% on average. We can conclude that the sequential porting of NPB-CPP is reliable and efficient compared to the NPB in different compilers and platforms.

4.2. Parallel porting

This section aims at comparing the performance of the parallel porting on shared-memory architectures, which was first implemented with OpenMP in Fortran by the NAS experts. The OpenMP version in C++ follows the parallel design implementation described in Section 3.2.1. Consequently, we are comparing the behavior of OpenMP parallelism between NPB-CPP and NPB benchmarks. Graphs in Fig. 6 show the execution time in seconds (using a logarithmic scale), varying the number of threads from 1 to 48 on the Xeon platform and with workload Class C. The plots have a second Y-axis to report the normalized difference in percentage between NPB-CPP and the NPB using bars. A positive difference means that the NPB-CPP version is faster than the NPB, negative values vice-versa.

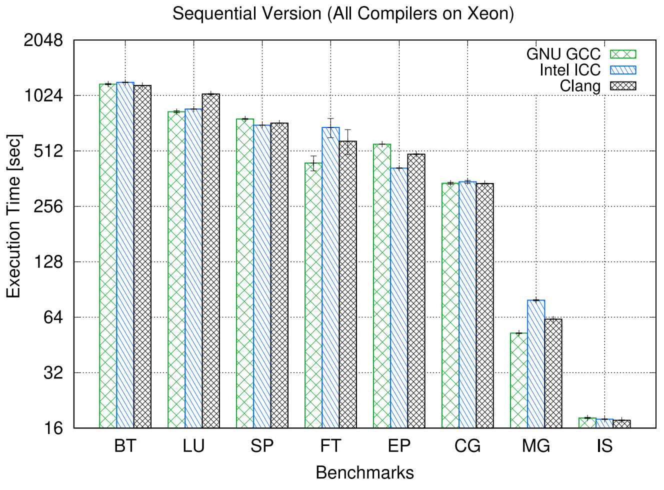


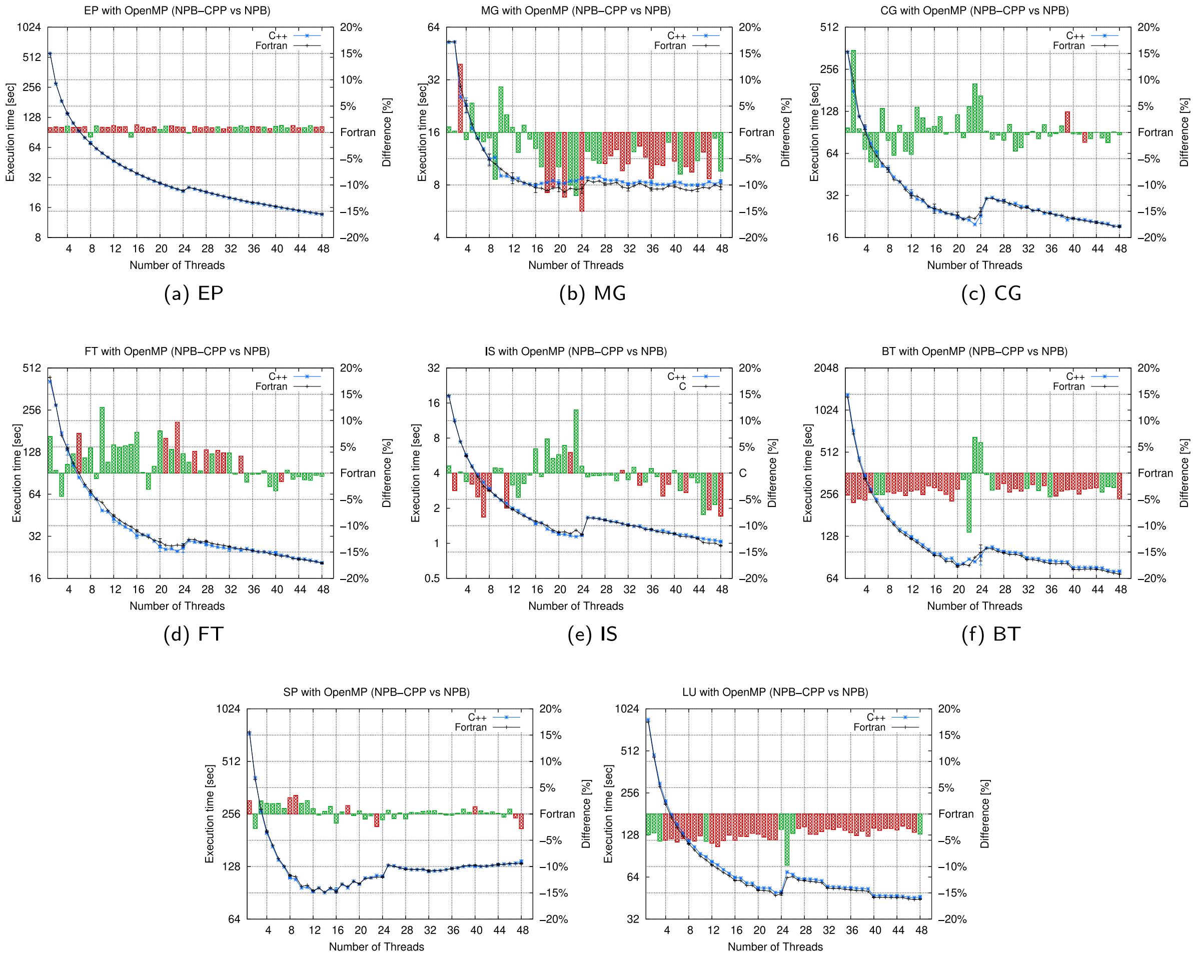
Fig. 5. Comparing the performance of the NPB-CPP benchmarks on the Xeon platform using all compilers and Class C.

To better evaluate the differences for each parallel execution, we colored the bars with red and green colors. The assigned color is the result obtained by the p-value statistical analysis [43]. The smaller the p-value, the stronger the evidence that the null hypothesis should be rejected. Before running the statistical test, we ran the homogeneity test to evaluate if the sample (5 repetitions) is in a normal distribution using the Shapiro–Wilk test. This test allows to identify, even with a small sample, which kind of hypothesis test to perform (parametric or non-parametric). We ran the paired T-test when both samples of Fortran and C++ were normal distributed. Otherwise, we ran the paired Wilcoxon test. In our statistical analysis, the null hypothesis (H_0) is NPB=NPB-CPP (e.g., the NPB-CPP provides the same level of parallel performance of the original NPB). The alternative hypothesis (H_1) is then NPB$\neq$NPB-CPP. To reject H_0, the p-value must be less than 0.05 (this is a commonly used threshold in the literature for software experiments). When H_0 (green color) is rejected, we assume H_1 (red color). In this way, it is possible to quickly identify which results are statistically different considering 95% confidence in Fig. 6.

As sketched in Fig. 6, the performance among NPB and NPB-CPP benchmarks are very close. The EP and MG kernels (Figs. 6(a) and 6(b)), followed by BT and LU pseudo-applications (Figs. 6(f) and 6(h)), are the benchmarks presenting more cases with significant statistical difference. For LU the primary reason is that the NPB-CPP sequential version is, on average, 4.3% slower than the NPB version. This difference persists in the parallel version (Fig. 6(h)). In BT, while in the NPB-CPP sequential version was 7.43% faster, the parallel version shows that NPB-CPP with OpenMP configured with one thread is 4.15% slower than NPB, which is in Fortran. The BT and SP pseudo-applications are both bounded to the sequential PDE solver. However, in the Xeon platform, only SP stops scaling around 12 threads, which explains why its results tend to diverge less.

FT and MG benchmarks presented larger performance differences. The first in favor of the NPB-CPP version whereas the second in favor of the NPB version. Something similar happens to CG, where results are different since it executes over irregular workloads that require memory locality. For IS, on average, the execution time is similar except for the low and high parallelism degree. Finally, in the EP kernel, NPB-CPP is slightly faster than NPB in all cases.

To observe how the performance is impacted by the use of different compilers, Table 4 summarizes the best parallel execution times obtained on the Xeon platform using the gcc, icc,



(a) EP (b) MG (c) CG

(d) FT (e) IS (f) BT

(g) SP (h) LU

Fig. 6. Experimental results for NPB-CPP and NPB benchmarks parallelized with OpenMP on the `Xeon` platform by using the GNU compiler and workload Class C. Lines show the execution time in seconds (using log scale) while bars show the normalized difference. Bar's colors show if this difference is significant from the statistical standpoint under 95% of reliability: green color means `NPB=NPB-CPP` whereas red means `NPB≠NPB-CPP`.

and `clang` compilers with workload Class C. For `gcc` and `icc`, on average the results are close to each other, except for EP, FT, and MG. EP executes 25.6% faster with `icc` than `gcc`. Both FT and MG stress long-distance data communications and their performance relies mainly on how much the compiler is capable of optimizing memory accesses. The explanation of MG and FT also applies to `clang`. Besides, `clang` results are varying for the three pseudo-applications. In LU, the sequential execution time is almost 25% higher for `clang` than the others. However, in BT and SP `clang` optimizes better the PDE solvers and achieves better results for similar degrees of parallelism.

After presenting and discussing these experiments, we can conclude that the parallel porting achieved reliable results and efficient performance compared to the original version. The statistical analysis revealed that 58.8% of the tests (combining the number of threads and benchmarks) stand equal for NPB and NPB-CPP. In addition to that, we can see that the differences were small in average. The next section will extend the parallelism performance analysis on other parallel programming frameworks.

4.3. NPB-CPP with Intel TBB and FastFlow

As the previous experiments highlighted the reliability and efficiency of the sequential and parallel porting, this section will evaluate and discuss the performance scalability outcomes of the benchmarks for other parallel programming frameworks, which now can be tested on these benchmarks due to our work. Therefore, we used as baseline the performance results of NPB-CPP with OpenMP (previously discussed) to compare with the NPB-CPP code parallelized using Intel Threading Building Blocks (TBB) and FastFlow (FF) (see Section 3.2). Since our experimental environment uses NUMA architectures, thread affinity can significantly affect performance. We used the default configuration for each parallel programming framework. Therefore, OpenMP and Intel TBB let the OS handle thread affinity while FastFlow places threads to physical cores first from 0 to N in ascending order.

The experiments were executed using all the three multicore platforms described in Table 3. The methodology is similar to the one previously described in Section 4.2. Therefore, we executed 5 repetition for each configuration from 1 to maximum degree of parallelism in each platform. The results present the best average execution time in seconds and the standard deviation. The table shows the degree of parallelism where we achieve the best speedup. The speedup is calculated using the ratio between NPB-CPP's sequential version and the best execution time achieved for each parallel programming framework. To investigate the behavior of the benchmarks, platforms, and parallel frameworks on different classes we used the workloads from Classes B and C. Table 5 reports the best speedups of each version computed over the NPB-CPP sequential version, the best execution time in seconds, the number of threads used, and the standard deviation. These metrics are presented for the three platforms.

Table 4
The best execution times among NPB and NPB-CPP benchmarks on the Xeon platform by using gcc/icc/clang compilers with workload Class C on OpenMP versions.

Bench.	Metrics	GNU GCC		Intel ICC		clang
		C++	Fortran	C++	Fortran	C++
EP	N Threads Time (s) Std. Dev.	48 13.60 0.01	48 13.75 0.05	48 10.40 0.03	48 10.35 0.05	48 12.99 0.05
MG	N Threads Time (s) Std. Dev.	16 7.94 0.12	21 7.28 0.24	41 8.35 0.17	22 8.02 0.24	22 7.66 0.39
CG	N Threads Time (s) Std. Dev.	48 19.27 0.18	48 19.18 0.34	47 19.32 0.67	24 18.29 0.91	45 20.02 0.57
FT	N Threads Time (s) Std. Dev.	48 20.67 0.08	48 20.54 0.23	48 22.11 0.99	47 23.71 0.40	47 23.38 0.86
IS	N Threads Time (s) Std. Dev.	48 1.03 0.01	48 0.96 0.01	47 1.02 0.01	47 1.08 0.01	47 1.00 0.03
BT	N Threads Time (s) Std. Dev.	47 71.74 0.59	48 68.61 0.23	24 71.96 0.89	23 66.16 0.53	23 64.18 0.67
SP	N Threads Time (s) Std. Dev.	14 90.75 1.49	14 91.20 0.69	14 90.05 0.58	14 88.86 0.74	14 85.92 0.44
LU	N Threads Time (s) Std. Dev.	47 45.72 0.29	47 44.18 0.38	45 45.72 0.22	45 44.88 0.21	40 57.52 0.12

EP is an embarrassingly parallel computation implemented by a single *MapReduce* pattern. As expected, it reaches its best speedup by using all the available cores of the machines in all parallel programming tools and platforms. TBB exhibits the highest speedup due to its work-stealing scheduling. In Class B, FastFlow is slightly faster because TBB's runtime overhead is more evident for the small workload. The MG kernel requires access to non-linear addresses in memory, preventing the best cache utilization. Since Power8 has a higher memory bandwidth [44], it provides better results. FF and OMP provide higher speedup, mainly due to the static scheduling policy of loop iterations adopted. The main difference between OMP and FF is that OMP creates a parallel region where threads are always active while FF disables and enables them each time a new parallel *Map* is executed.

The CG kernel requires a large number of synchronizations, primarily because it uses many *MapReduce* for sharing partial results (see Table 2). This periodically interrupts the computation and decreases the maximum performance achievable. CG has irregular data accesses, which benefit of bigger last-level caches shared between more cores. This explains why the results are better on the Xeon and Power8 platforms. The OMP version exhibits the highest speedups, mainly because we were able to add nowait directives to remove implicit barriers. Between the other two, TBB achieves better results than FF because it balances better the irregular workload using a work-stealing scheduler, while FF uses a static round-robin assignment.

In the FT kernel, data communication phases have a significant impact on performance. Each FFT resolution copies a slice of data from the main 3D-matrix to a local 1D-array, solves it, and then copies it back into the main matrix. In the Xeon platform, results are similar for all versions because the memory bottleneck hides other sources of overhead. However, in the Epyc and Power8 platforms, although speedups are similar, the obtained sequential execution time in the Epyc machine is an order of magnitude faster than the one obtained in the Power8. This explains the better scaling in the Power8, specifically for the TBB version. It benefits from its dynamic work-stealing scheduling policy, which achieves a good workload balancing among the threads. However, since the computation in FT is fine grained, the extra overhead of such dynamic scheduling does not payback on the Epyc platform. As IS kernel is also memory bound, it explains the higher speedups in Power8. The IS sequential code runs faster in Epyc than in Power8. This is the reason why FF and TBB obtain the best speedup earlier than OMP as the execution time is short (in the order of seconds) and the runtime overhead has a higher impact. In the Power8, FF's thread pinning to cores mechanism showed poor performance for this benchmark.

As already discussed, BT and SP use analogous PDE solvers. Despite both are bound by the PDE solver, taking into account that they execute a different factorization, the bottleneck occurs in different situations. This explains why BT has a higher speedup than SP and why they stop scaling when the sequential bottleneck of the solver is reached. For SP, we parallelized it by using the same strategy for all parallel programming frameworks, implementing only *Map*s. Therefore, the maximum speedup is very close in all implementations. In Class C, TBB achieves the best speedups for Epyc and Power8 because it balances better the workload. However, the small workload of Class B swaps the results and OMP obtains the better speedups mainly due to the smaller runtime overhead. For BT, the PDE solvers bottleneck is evident in the Epyc platform while not in Xeon and Power8. Concerning the runtime systems of the parallel programming frameworks, the static iteration scheduling employed by the OMP and FF versions provide better speedup with respect to the dynamic scheduling used by the TBB version. Specifically, the execution time is reduced significantly if the iteration space is equally divided among all threads. In the Power8 platform, this happens with 160 threads.

The LU pseudo-application has been parallelized by using an implicit multidimensional pipeline implemented with *Maps*. It uses the static scheduling policy for the iteration space and proper lock mechanisms for synchronizations. In all platforms, the maximum speedup is low because the benchmark is limited by the sequential resolution of the linear system of equations. For FF and TBB, the execution flow management is done in the thread scope while for OMP it is done inside the loop iteration scope.

In summary, we conclude that different parallel programming frameworks can be effective on NPB-CPP benchmarks by using equivalent pattern-based implementations, achieving good performance as shown in Table 5. The results highlight that the performance of different parallel programming frameworks depends on the shared-memory architecture design and application characteristics. OpenMP is considered the *de-facto* standard framework for these environments, however, it does not always achieve the best speedups. These insights open space for future investigations regarding parallelism optimizations.

4.4. Comparison with our prior work

In this section, we provide the performance comparison between NPB-CPP and our prior work [24] which, as already explained in Section 1, was based on the five kernels only and on an outdated version of NPB.

In terms of parallel implementation, the IS and FT kernels exhibit the most significant differences between the new code in NPB-CPP and the old one. In IS, we implement a new parallel strategy for FF and TBB designed from scratch. This strategy uses a single Map for parallelizing the complete compute-intensive region of the sorting routine using buckets. In contrast, the previous one used four Maps and serialized the synchronization between buckets, generating extra overhead. Furthermore, the

Table 5
Experimental results for NPB-CPP showing the best execution time (in seconds) and speedup using Class B and Class C for Xeon, Epyc, and Power8. The speedup is calculated using the ratio between NPB-CPP's sequential version (no parallel framework) and the best execution time for each tool. Colored cells highlight the winner speedup.

Bench.	Metrics	Xeon						Epyc						Power8					
		Class B			Class C			Class B			Class C			Class B			Class C		
		TBB	FF	OMP	TBB	FF	OMP	TBB	FF	OMP	TBB	FF	OMP	TBB	FF	OMP	TBB	FF	OMP
EP	N Threads	48	47	48	48	48	48	128	128	120	128	128	124	152	160	148	160	160	160
	Time (s)	3.39	3.78	3.42	13.52	15.04	13.60	1.26	1.25	1.42	4.92	5.02	5.36	2.24	2.15	2.41	8.58	10.27	8.89
	Speedup	41.10	36.90	40.08	41.20	37.02	40.94	120.05	120.24	106.67	128.77	126.35	118.38	62.06	64.71	57.68	124.03	103.60	119.74
	Std. Dev.	0.00	0.04	0.01	0.01	0.11	0.01	0.02	0.00	0.04	0.01	0.01	0.11	0.10	0.01	0.11	0.06	0.26	0.29
MG	N Threads	10	16	14	19	16	16	8	9	8	8	9	16	5	4	6	64	156	64
	Time (s)	1.34	0.89	0.87	10.18	7.69	7.94	1.82	1.39	1.26	14.76	15.28	13.64	1.43	1.96	1.40	18.28	20.73	15.83
	Speedup	4.27	6.40	6.58	5.15	6.82	6.60	2.19	2.88	3.17	2.25	2.17	2.43	3.16	2.30	3.23	16.51	14.55	19.05
	Std. Dev.	0.02	0.04	0.01	0.03	0.18	0.12	0.03	0.08	0.06	0.57	1.36	1.10	0.05	0.03	0.03	0.14	0.14	0.02
CG	N Threads	35	23	48	48	48	48	40	6	40	56	6	24	72	68	112	152	68	120
	Time (s)	7.84	9.16	6.86	20.11	20.52	19.27	14.16	14.91	14.56	37.03	38.88	35.62	9.80	12.18	8.66	34.80	35.19	30.47
	Speedup	14.95	12.80	17.09	16.99	16.65	17.73	2.99	2.84	2.91	3.09	2.94	3.21	10.02	8.06	11.34	28.81	28.49	32.90
	Std. Dev.	0.04	0.15	0.01	0.08	0.52	0.18	0.11	0.34	1.65	0.52	1.31	4.66	0.17	0.14	0.43	0.31	0.36	0.32
FT	N Threads	45	48	48	48	48	48	124	124	128	56	124	128	152	128	140	160	152	140
	Time (s)	5.21	4.71	4.80	20.62	20.06	20.67	3.02	2.86	2.79	12.00	11.42	11.00	4.65	4.90	4.71	20.60	23.51	25.10
	Speedup	18.18	20.12	19.71	21.32	21.92	21.28	40.86	43.06	44.11	64.44	67.71	70.23	14.54	13.80	14.36	72.70	63.73	59.72
	Std. Dev.	0.04	0.02	0.10	0.05	0.11	0.08	0.01	0.05	0.03	0.08	0.19	0.16	0.05	0.01	0.03	0.10	0.25	0.31
IS	N Threads	46	47	48	48	47	48	48	84	128	28	24	124	52	156	88	140	160	148
	Time (s)	0.25	0.25	0.21	1.21	1.06	1.03	0.32	0.3	0.214	1.32	1.16	0.87	0.26	0.23	0.22	1.09	1.27	0.98
	Speedup	17.67	17.67	20.83	14.95	17.06	17.56	14.27	15.23	21.35	13.97	15.94	21.16	6.26	7.31	7.50	43.81	37.6	48.53
	Std. Dev.	0.00	0.00	0.01	0.02	0.03	0.01	0.01	0.00	0.00	0.03	0.08	0.02	0.01	0.01	0.01	0.02	0.03	0.04
BT	N Threads	46	47	20	41	46	47	60	100	104	52	104	100	36	100	108	160	160	160
	Time (s)	22.56	17.44	18.70	86.31	68.15	71.74	13.96	13.14	12.27	60.70	60.34	59.63	18.99	18.46	15.54	210.73	105.69	86.60
	Speedup	12.27	15.88	14.81	13.67	17.32	16.45	13.62	14.47	15.49	12.84	12.92	13.07	13.77	14.17	16.84	29.05	57.93	70.70
	Std. Dev.	0.13	0.04	0.17	0.36	0.31	0.59	0.09	0.24	0.33	0.25	1.53	1.11	0.23	0.16	0.58	0.75	2.02	0.78
SP	N Threads	22	22	20	13	14	14	20	9	11	8	7	12	20	12	12	40	32	32
	Time (s)	18.85	17.18	16.26	97.69	93.35	90.75	27.27	22.06	21.05	107.81	113.49	112.21	19.89	22.49	18.94	166.52	172.85	171.95
	Speedup	9.69	10.63	11.23	7.82	8.18	8.41	4.13	5.10	5.34	4.14	3.94	3.98	7.61	6.73	7.99	16.78	16.16	16.25
	Std. Dev.	0.18	0.13	0.08	1.22	2.65	1.49	0.21	0.77	0.83	1.92	7.48	13.41	0.56	0.06	0.29	0.31	1.30	3.56
LU	N Threads	35	38	47	41	46	47	20	28	36	28	24	32	12	52	104	80	80	80
	Time (s)	13.85	12.50	12.45	47.96	43.64	45.72	13.26	12.07	10.23	47.21	47.89	47.21	23.68	22.95	12.71	85.89	88.24	82.92
	Speedup	14.37	15.91	15.98	17.42	19.14	18.27	9.35	10.28	12.13	11.45	11.29	11.45	7.09	7.31	13.20	34.08	33.17	35.30
	Std. Dev.	0.07	0.03	0.15	0.23	0.15	0.29	0.20	0.22	0.37	0.58	0.80	2.60	1.28	0.02	0.37	0.35	1.75	0.97

previous implementation was not implemented in TBB due to issues related to getting the thread identifiers, which was not possible with task-based parallelism for TBB. In the FT kernel, we modified some routines such as the `evolve` and `checksum`, and included others like the initialization for warming up all data before execution. We identified an extra loop suitable to be parallelized using a Map. Additionally, in the CG kernel we identified four extra Maps that provide performance benefits with large problem sizes (starting from class C).

Table 6 shows the performance improvements between our two versions. Since the code in prior work does not work on large problem sizes starting from class C, we restricted the comparison with experiments using class B. Considering this workload size is relatively small (see [42]), we do not expect large performance differences. The table presents the best execution times for the five kernels on the Xeon platform with the GNU compiler. For the EP kernel, the implementation in NPB-CPP is faster than the prior one in all the considered parallel programming frameworks, more distinct in FF. The main reason is that our previous implementation was affected by false-sharing problems, which we have fixed in the NPB-CPP implementation using proper padding of our data structures and arrays (configured to automatically work in any multicore platform by reading the cache line sizes from the operating system). Also in the MG kernel, the new version is faster in all the tools. The main reason is that we standardized the implementation of linear arrays to all kernels, which work better in this case for MG's fine-grained workload and irregular memory accesses. In the old implementation we used multidimensional arrays.

In CG, FF and TBB had different performance result because we now parallelized more loops than before. In FF, we used a different scheduler than the one used in the other tools, because it works better with its thread pinning mechanism in CG. However, it payoffs with bigger workloads as shown in Table 5, where FF is similar to the others. TBB also shows a higher overhead than OMP in CG. However, TBB overhead is much more evident in FT since it is the only tool configured to use a dynamic scheduler (work-stealing). Additionally, in FT we modified considerably the code, and consequently, all the new versions are faster. The IS execution time is very small, close to 250 ms, and the overhead of the parallel runtime becomes more evident. The difference between the FF versions is close to 5%, which depends on the new parallel strategy that paybacks even more with larger workload sizes. Finally, this new version allows the Intel TBB parallelization of IS, not available before.

Table 6
Comparison with our prior work: best execution times on Xeon with Class B using the GNU compiler.

Bench.	Metrics	Current work			Previous work [24]		
		FF	OMP	TBB	FF	OMP	TBB
EP	N Threads	47	48	48	48	48	48
	Time (s)	3.78	3.42	3.39	5.07	3.45	3.41
	Std. Dev.	0.05	0.01	0.00	0.03	0.00	0.01
MG	N Threads	16	14	10	21	16	9
	Time (s)	0.89	0.87	1.34	1.01	0.90	1.40
	Std. Dev.	0.04	0.01	0.02	0.02	0.00	0.06
CG	N Threads	23	48	35	48	48	47
	Time (s)	9.16	6.86	7.84	7.94	6.76	9.04
	Std. Dev.	0.15	0.01	0.04	0.03	0.14	0.07
FT	N Threads	48	48	45	48	47	48
	Time (s)	4.71	4.80	5.21	4.99	5.43	5.36
	Std. Dev.	0.02	0.11	0.04	0.06	0.11	0.06
IS	N Threads	48	48	48	48	48	–
	Time (s)	0.25	0.21	0.25	0.26	0.21	–
	Std. Dev.	0.00	0.00	0.00	0.00	0.00	–

5. Final remarks

In this section, we present a summary of our findings. For most benchmarks, the Map and MapReduce patterns exhibited the required semantics for expressing parallelism in the code. Only in the IS kernel and the LU pseudo-application we introduce a small alteration of the pattern semantics. Consequently, these two benchmarks contain extra hand-written synchronization strategies that are not abstracted by the Map or MapReduce, and the application programmer must provide them manually. In IS, although not ideal and error-prone, introducing parallelism is easier because NPB's sequential version already implements the entire routine for synchronizing neighbors after parallel executions. In OpenMP this routine can be used along with the `parallel` and `for` pragma directives, which are executed independently. However, in TBB and FastFlow we must adjust the Map and MapReduce pattern semantics to overcome their API limitations (see 3.2.6), while in OpenMP it remains the same. For the LU pseudo-application, the same happens. However, achieving parallelism is more difficult because synchronization is needed between parallel instances during execution time. Unfortunately, the Map and MapReduce patterns lack expressiveness, and a programmer must manually implement the spinning lock mechanism for synchronizing the data flow. This parallel pattern semantic problem applies to all the three frameworks considered in our work.

Another high-level semantic limitation we observed concerns the MapReduce reduction operation. The problem relates with the lack of expressiveness for implementing Reduce routines. Specifically, the default MapReduce pattern abstractions provided by the existing frameworks are not ideal for modeling common reduction operations. For example, in many NPB benchmarks there are reductions executed over dynamic arrays and custom data types. Using the features available in the frameworks turn the Reduce implementation non-trivial, again engaging the application programming with low-level parallelism.

In terms of performance, Table 5 presented a summary of our findings regarding the low-level pattern implementations. Problems and limitations in the parallel pattern implementation semantics are directly connected to performance drawbacks. Although in TBB and FastFlow the Map adjustments were required, OpenMP achieved better performance. In LU, OpenMP did not achieved the best results in one situation because it relies in a coarser-grained lock strategy than TBB and FastFlow. Furthermore, the removing of implicit barriers is only available for OpenMP, which due to that achieves better results. However, TBB has a better load balancing mechanism, achieving linear speedup in EP and better scaling in other benchmarks.

As shown in Table 5, we discover that many benchmarks do not scale well. Although this derives from several implementation and architectural reasons, such as cache misses, floating-point unit, and memory bandwidth constraints, some performance drawbacks are also related to the Map and MapReduce patterns themselves. We highlight two major drawbacks that should be considered for further research: (1) in many situations the overhead introduced by the Map pattern is higher than the benefit of exploiting such fine-grained parallelism; (2) the thread mapping lacks data flow awareness. OpenMP and TBB let the OS to choose where threads will be allocated while FastFlow provides a default and custom one. Contrasting the overall results, there is room for improving thread mapping.

6. Conclusions

This paper provided the NPB-CPP benchmark suite with parallel implementations following a structured parallel programming approach and using OpenMP, Intel TBB, and FastFlow for shared-memory architectures. To the best of our knowledge, this work is the first attempt to provide equivalent pattern-based implementations among different parallel programming frameworks in C++ using the NPB suite. Furthermore, we presented a comprehensive discussion about the benchmarks, the implementation, and also our parallel design choices for justifying all the results. For evaluating the C++ parallel programming frameworks, we first obtained a C++ version of the NPB, named NPB-CPP. Moreover, we characterized the basic structure of the benchmarks and their distinguished importance in the research, which relies on the fact that NPB contains many irregular workloads stressing different components of the underlying multicore architecture.

We evaluated NPB-CPP by performing a set of experiments using popular platforms nowadays (`Xeon`, `Epyc`, and `Power8`) with three different compilers (GNU GCC, Intel ICC, and Clang). The performance evaluation was guided using statistical analysis. The results demonstrated that there are minor performance differences among the sequential versions. OpenMP versions (NPB vs NPB-CPP) have shown a reliable outcome, presenting a similar pattern of behavior. Additionally, with 95% of confidence, the statistical analysis evaluating the parallel porting revealed that, at least, 58.8% of the samples stand for NPB-CPP not significant different from the NPB benchmarks.

Finally, NPB-CPP's extensibility and its importance to the scientific community were shown in the experiments of the parallel versions using FastFlow and TBB. Other related abstractions can be evaluated, including those that were designed for distributed and heterogeneous parallel architectures. The experiments, using `Xeon`, `Epyc`, and `Power8` revealed that the performance of the kernels and pseudo-applications varies among the platforms, benchmarks, and parallel frameworks. The main performance bottleneck in the benchmarks was due to the caches and memory bandwidth among the platforms, resulting in different behaviors between the parallel programming frameworks.

Nonetheless, our experiments highlight some trends to help other programmers choosing the best approach depending on the workload. FastFlow obtained better performance for `Xeon` when comparing among other platforms. Intel TBB obtained better performance in a few benchmarks and platforms mainly due to its optimized scheduling for balancing the irregular workload among threads. Although OpenMP is considered the state-of-the-art for multicores, this work revealed many situations in which TBB and FastFlow are more optimized and can perform better.

Researchers can use NPB-CPP to improve parallel programming abstractions based on the insights already revealed in this research. Possible future works are: (1) provide NPB-CPP with other parallel programming frameworks for shared, heterogeneous, and distributed memory architectures; (2) assess the performance of NPB-CPP by providing different parallel patterns or compositions of them; (3) perform experiments on other related multicore architectures such as ARM processors; and (4) assess compilers and new code optimizations techniques.

CRediT authorship contribution statement

Júnior Löff: Software, Investigation, Validation, Visualization, Writing – original draft. **Dalvan Griebler:** Conceptualization, Formal analysis, Project administration, Validation, Writing – review & editing. **Gabriele Mencagli:** Conceptualization, Writing – review & editing. **Gabriell Araujo:** Software, Validation, Writing – original draft. **Massimo Torquati:** Conceptualization, Validation, Resources, Writing – review & editing. **Marco Danelutto:** Resources, Writing – review & editing. **Luiz Gustavo Fernandes:** Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research is partially funded by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001, FAPERGS 01/2017-ARD project ParaElastic (N° 17/2551-0000871-5), FAPERGS 05/2019-PQG project ParAS (N° 19/2551-0001895-9), FAPERGS 10/2020-ARD project SPar4.0 (N° 21/2551-0000725-7), and Universal MCTIC/CNPq N° 28/2018 project SParCloud (No. 437693/2018-0).

References

[1] B. Chapman, G. Jost, R. van der Pas, Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation), MIT Press, London, UK, 2007.

[2] M. Voss, R. Asenjo, J. Reinders, Pro TBB: C++ Parallel Programming with Threading Building Blocks, first ed., Apress, USA, 2019.

[3] OpenMP Architecture Review Board, OpenMP Application program interface version 5.0, 2018, URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[4] H. González-Vélez, M. Leyton, A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers, Softw. - Pract. Exp. 40 (12) (2010) 1135–1160, http://dx.doi.org/10.1002/spe.1026.

[5] M. Danelutto, G. Mencagli, M. Torquati, H. González-Vélez, P. Kilpatrick, Algorithmic skeletons and parallel design patterns in mainstream parallel programming, Int. J. Parallel Program. (2020) http://dx.doi.org/10.1007/s10766-020-00684-w.

[6] T. Mattson, B. Sanders, B. Massingill, Patterns for Parallel Programming, first ed., Addison-Wesley Professional, 2004.

[7] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Torquati, Fastflow: High-level and efficient streaming on multicore, in: Programming Multi-Core and Many-Core Computing Systems, John Wiley & Sons, Ltd, 2017, pp. 261–280, http://dx.doi.org/10.1002/9781119332015.ch13.

[8] A. Ernstsson, L. Li, C. Kessler, SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems, Int. J. Parallel Program. 46 (1) (2018) 62–80, http://dx.doi.org/10.1007/s10766-017-0490-5.

[9] C. Bienia, S. Kumar, J.P. Singh, K. Li, The PARSEC benchmark suite: Characterization and architectural implications, in: Proceedings of the 17 International Conference on Parallel Architectures and Compilation Techniques, in: PACT '08, ACM, 2008, pp. 72–81.

[10] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta, The SPLASH-2 programs: Characterization and methodological considerations, SIGARCH Comput. Archit. News 23 (2) (1995) 24–36.

[11] J.L. Henning, SPEC CPU2006 benchmark descriptions, SIGARCH Comput. Archit. News 34 (4) (2006) 1–17, http://dx.doi.org/10.1145/1186736.1186737.

[12] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, S. Weeratunga, The NAS Parallel Benchmarks, Technical Report, NASA Ames Research Center, Moffett Field, CA - USA, 1994.

[13] D. De Sensi, T. De Matteis, M. Torquati, G. Mencagli, M. Danelutto, Bringing parallel patterns out of the corner: The P3ARSEC benchmark suite, ACM Trans. Archit. Code Optim. 14 (4) (2017) 33:1–33:26.

[14] K. Ejjaaouani, O. Aumage, J. Bigot, M. Mehrenberger, H. Murai, M. Nakao, M. Sato, INKS, a programming model to decouple performance from algorithm in HPC codes, in: Euro-Par 2018: Parallel Processing Workshops, Springer International Publishing, Cham, 2019, pp. 757–768.

[15] A.M. Garcia, M. Serpa, D. Griebler, C. Schepke, L.G. Fernandes, P. Navaux, The impact of CPU frequency scaling on power consumption of computing infrastructures, in: Computational Science and Its Applications – ICCSA 2020, Springer International Publishing, Cham, 2020, pp. 142–157.

[16] A. Handleman, A.G. Rattew, I.-T.A. Lee, T.B. Schardl, A hybrid scheduling scheme for parallel loops, in: 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2012, pp. 587–598, http://dx.doi.org/10.1109/IPDPS49936.2021.00067.

[17] J. ao Vicente Ferreira Lima, I.R. s, L. Lefèvre, T. Gautier, Performance and energy analysis of OpenMP runtime systems with dense linear algebra algorithms, Int. J. High Perform. Comput. Appl. 33 (3) (2019) 431–443, http://dx.doi.org/10.1177/1094342018792079.

[18] A. M. Garcia, C. Schepke, A. Girardi, PAMPAR: A new parallel benchmark for performance and energy consumption evaluation, Concurr. Comput.: Pract. Exper. 32 (20) (2020) e5504, http://dx.doi.org/10.1002/cpe.5504.

[19] N. Mietzsch, K. Fuerlinger, Investigating performance and potential of the parallel STL using NAS parallel benchmark kernels, in: 2019 International Conference on High Performance Computing Simulation (HPCS), 2019, pp. 136–144, http://dx.doi.org/10.1109/HPCS48598.2019.9188147.

[20] S. Prema, R. Nasre, R. Jehadeesan, B. Panigrahi, A study on popular auto-parallelization frameworks, Concurr. Comput.: Pract. Exper. 31 (17) (2019).

[21] G.A. de Araujo, D. Griebler, M. Danelutto, L.G. Fernandes, Efficient NAS parallel benchmark kernels with CUDA, in: 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2020, pp. 9–16, http://dx.doi.org/10.1109/PDP50117.2020.00009.

[22] A.M. Maliszewski, D. Griebler, C. Schepke, A. Ditter, D. Fey, L.G. Fernandes, The NAS benchmark kernels for single and multi-tenant cloud instances with LXC/KVM, in: 2018 International Conference on High Performance Computing Simulation (HPCS), 2018, pp. 359–366, http://dx.doi.org/10.1109/HPCS.2018.00066.

[23] M. McCool, A. Robison, J. Reinders, Structured Parallel Programming: Patterns for Efficient Computation, Elsevier Science, 2012.

[24] D. Griebler, J. Löff, G. Mencagli, M. Danelutto, L.G. Fernandes, Efficient NAS benchmark kernels with C++ parallel programming, in: 26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2018, pp. 733–740.

[25] H.-Q. Jin, M. Frumkin, J. Yan, The OpenMP Implementation of NAS Parallel Benchmarks and its Performance, Technical Report, NASA Ames Research Center, Moffett Field, CA, USA, 1999.

[26] S. Seo, G. Jo, J. Lee, Performance characterization of the NAS parallel benchmarks in OpenCL, in: 2011 IEEE International Symposium on Workload Characterization (IISWC), 2011, pp. 137–148, http://dx.doi.org/10.1109/IISWC.2011.6114174.

[27] Y. Do, H. Kim, P. Oh, D. Park, J. Lee, SNU-NPB 2019: Parallelizing and optimizing NPB in OpenCL and CUDA for modern GPUs, in: 2019 IEEE International Symposium on Workload Characterization (IISWC), 2019, pp. 93–105.

[28] M. Popov, C. Akel, F. Conti, W. Jalby, P. de Oliveira Castro, PCERE: Fine-grained parallel benchmark decomposition for scalability prediction, in: 2015 IEEE International Parallel and Distributed Processing Symposium, 2015, pp. 1151–1160.

[29] Omni Compiler, Omni compiler project, 2017, URL: http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/.

[30] T. El-Ghazawi, F. Cantonnet, UPC performance and potential: A NPB experimental study, in: SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, 2002, p. 17.

[31] K. Yelick, Berkeley UPC - Unified parallel C, 2019, URL: https://upc.lbl.gov/.

[32] H.-Q. Jin, R. Hood, P. Mehrotra, A practical study of UPC using the NAS parallel benchmarks, in: Proceedings of the Third Conference on Partitioned Global Address Space Programing Models, PGAS '09, 2009, pp. 8:1–7.

[33] H.-N. Vießmann, A. Šinkarovs, S.-B. Scholz, Extended memory reuse an optimisation for reducing memory allocations, in: ACM International Conference Proceeding, 2018, pp. 107–118.

[34] G. Cong, H. Wen, H. Murata, Y. Negishi, Tool-assisted optimization of shared-memory accesses in UPC applications, in: 2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems, 2012, pp. 104–111.

[35] A. Anbar, A.-H.A. Badawy, O. Serres, T. El-Ghazawi, Where should the threads go? Leveraging hierarchical data locality to solve the thread affinity dilemma, in: Proceedings of the International Conference on Parallel and Distributed Systems, 2015-April, 2014, pp. 384–391.

[36] O. Serres, A. Kayi, A. Anbar, T. El-Ghazawi, Enabling PGAS productivity with hardware support for shared address mapping: A UPC case study, ACM Trans. Archit. Code Optim. 12 (4) (2015).

[37] J. Diaz, C. Munoz-Caro, A. Nino, A survey of parallel programming models and tools in the multi and many-core era, IEEE Trans. Parallel Distrib. Syst. 23 (8) (2012) 1369–1386.

[38] A. Trotman, M. Crane, Micro- and macro-optimizations of SaaT search, Softw. - Pract. Exp. 49 (5) (2019) 942–950, http://dx.doi.org/10.1002/spe.2683.

[39] D. Griebler, M. Danelutto, M. Torquati, L.G. Fernandes, SPar: A DSL for high-level and productive stream parallelism, Parallel Process. Lett. 27 (01) (2017) 20.

[40] G. Mencagli, M. Torquati, D. Griebler, M. Danelutto, L.G. Fernandes, Raising the parallel abstraction level for streaming analytics applications, IEEE Access 7 (2019) 131944–131961, http://dx.doi.org/10.1109/ACCESS.2019.2941183.

[41] D. del Rio Astorga, M.F. Dolz, J. Fernández, J.D. Garcí a, A generic parallel pattern interface for stream and data processing, Concurr. Comput.: Pract. Exper. 29 (24) (2017) e4175, http://dx.doi.org/10.1002/cpe.4175.

[42] NASA, The NAS parallel benchmarks, 2020, URL: https://www.nas.nasa.gov/publications/npb.html.

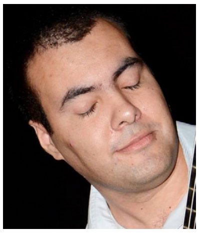
[43] D. Taeger, S. Kuhnt, Statistical Hypothesis Testing with SAS and R, first ed., Wiley Publishing, 2014.

[44] Clabby Analytics, Infrastructure Matters: POWER8 vs. Xeon x86, Technical Report, Clabby Analytics, 2014.

Júnior Löff is a M.Sc. student in Computer Science at the Pontifical Catholic University of Rio Grande do Sul (PUCRS), and research member of the Parallel Applications Modeling Group (GMAP) at PUCRS. He received his B.Sc. Degree in Computer Engineering from PUCRS in 2020. His research interests include: Parallel and distributed systems, high-performance applications modeling and hardware/software co-design.

Dalvan Griebler is an Associate Professor at the Pontifical Catholic University of Rio Grande do Sul (PUCRS). Also Associate Professor at Três de Maio Faculty (Setrem) and head of the Laboratory of Advanced Research on Cloud Computing (LARCC) at Setrem. He received the Ph.D. in computer science from both PUCRS and University of Pisa in 2016. His main research interests are: parallel and distributed computing, methodologies, languages and libraries for high-level parallel programming; benchmarking; and cloud computing.

Gabriele Mencagli is an Associate Professor at the Computer Science Department of the University of Pisa, Italy. He got his Ph.D. from the same University in 2012. He is co-author of more than 60 peer-reviewed papers appeared in international conferences, workshops and journals, and of one book. His research interests are in the area of Parallel and Distributed Systems, Autonomic Computing and Data Stream Processing. He is a member of the Editorial Board of Future Generation Computer Systems (Elsevier) and Cluster Computing (Springer).



Gabriell Araujo is a M.Sc. student in Computer Science at the Pontifical Catholic University of Rio Grande do Sul (PUCRS), and research member of the Parallel Applications Modeling Group (GMAP) at PUCRS. He received his B.Sc Degree in Computer Science, Music, and Entrepreneurship and Innovation from the Pontifical Catholic University of Rio Grande do Sul (PUCRS) and IPA Methodist University Center. His research interests are: Parallel algorithms and programming techniques for many-core architectures.

Massimo Torquati is an Assistant Professor at the Department of Computer Science, University of Pisa, Italy. He has published more than 100 peer-reviewed papers in conference proceedings and international journals, mostly in the fields of parallel and distributed programming and run-time systems for parallel computing platforms. He has been involved in several Italian, EU, and industry-supported research projects. He is the maintainer and the main developer of the FastFlow parallel programming framework.

Marco Danelutto is a Full Professor at the Computer Science Department of the University of Pisa, Italy. His main research interests are in the field of parallel programming models, in particular in the area of parallel design patterns and algorithmic skeletons. He is author of more than 150 papers appearing in refereed international journals and conferences. He is the group leader of the Parallel Programming Model group at the same university, involved in a number of national and EU funded projects (CoreGRID, GRIDcomp, ParaPhrase, REPARA, RePhrase).

Luiz Gustavo Fernandes is an Associate Professor of the graduate program in computer science (PPGCC) at the Pontifical Catholic University of Rio Grande do Sul (PUCRS). His primary research interests are Parallel and Distributed Computing, High Performance Applications Modeling, Green Computing and Parallel Programming Interfaces. Dr. Fernandes received his Ph.D. in Computer Science from the Institut National Polytechnique de Grenoble (France) in 2002. He currently leads the Parallel Applications Modeling Group (GMAP) at PUCRS.