Efficient NAS Benchmark Kernels with C++ Parallel Programming

Dalvan Griebler, Junior Loff, Luiz G. Fernandes
Pontifical Catholic University of Rio Grande do Sul (PUCRS),
Parallel Application Modeling Group (GMAP),
Porto Alegre – Brazil
Email: {dalvan.griebler,junior.loff}@acad.pucrs.br, luiz.fernandes@pucrs.br

Gabriele Mencagli, Marco Danelutto
University of Pisa (UNIPI),
Computer Science Department,
Pisa – Italy
Email: {mencagli,marcod}@di.unipi.it

***Abstract*—Benchmarking is a way to study the performance of new architectures and parallel programming frameworks. Well-established benchmark suites such as the NAS Parallel Benchmarks (NPB) comprise legacy codes that still lack portability to C++ language. As a consequence, a set of high-level and easy-to-use C++ parallel programming frameworks cannot be tested in NPB. Our goal is to describe a C++ porting of the NPB kernels and to analyze the performance achieved by different parallel implementations written using the Intel TBB, OpenMP and FastFlow frameworks for Multi-Cores. The experiments show an efficient code porting from Fortran to C++ and an efficient parallelization on average.**

***Index Terms*—Parallel Programming, NAS Benchmark, Performance Evaluation, Data Parallelism, FastFlow, OpenMP, TBB.**

I. Introduction

The heterogeneous landscape of parallel architectures and parallel programming frameworks motivates the use of representative benchmarking suites to characterize their performance and efficiency. Well-established suites like PARSEC [1], NAS Parallel Benchmarks (NPB) [2], SPLASH [3] and Rodinia [4] include micro computational kernels and real-world applications. They are for measuring the performance achieved by different machines, using different run-time environments for parallel computing.

In this paper, we focus on the five micro-kernels of NPB, which are behind of the implementation of three pseudo-applications taken from the computational aerodynamics domain [5], [2]. For many years, NPB has been considered an important suite to evaluate the behavior of massively parallel systems with realistic workloads. Originally, most of the kernels in the suite (four out of five) are available as Fortran legacy code. The parallel implementations are available for shared-memory and distributed-nothing machines, supporting the parallelism with OpenMP and MPI.

These benchmarks are also essential to evaluate parallel programming libraries, frameworks, API, and tools. Every software that aims to exploit the parallelism of the new landscape of parallel architecture requires parallel programming. In C++, for instance, there are libraries used to develop parallel software easily [6], [7]. Since they provide higher-level abstractions to the application programmer, their performance must be evaluated and checked. Therefore, it is relevant porting this consolidated benchmarks to other languages so that many of these languages' parallel programming abstractions can be tested and evaluated along with the new architectures. Thus, developers can decide when and which framework to use.

Our goal is to describe, in a comprehensive way, the porting of the five kernels from the NAS benchmark suite to C++ code as well as the kernels' parallelization with Intel TBB [8], OpenMP (translation only) [9], and FastFlow [10]. After, we aim at evaluating and comparing the performance of our implementations with the C++ parallel programming frameworks. Besides the implementation issues, this work has a valuable impact regarding research for the following reasons that also represent our major contributions:

- according to Sec. II, NPB has been ported in different languages. However, their porting are not made for C++ and for the purpose of evaluating parallel programming frameworks. In our paper, a complete porting for C++ of the NPB kernels is provided and evaluated.
- the performance reliability of our porting in C++ is discussed by showing that our sequential code has comparable performance with the original Fortran code;
- owing to our C++ porting, different C++ parallel programming frameworks and libraries (e.g., Intel TBB, OpenMP, and FastFlow) are used to provide parallel implementations of the kernels. Such versions are analyzed and their performance compared.

The rest of this paper is organized as follows. Sect. II gives a brief overview of the literature. Sect III describes the basic features of the NPB kernels. Sect. IV introduces our C++ porting and parallelization using various frameworks. Finally, Sect. V is devoted to the experimental part, and Sect. VI gives the conclusion of this work.

II. Related Work

Over the years, the NPB kernels have been widely utilized for testing hardware-level strategies or compiler-level optimization. However, there are a set of parallel programming interfaces written over the C/C++ language [10], [8], [11] that could be used in NPB.

In the literature, different domains utilized the NPB to evaluate its solutions. It emphasizes the NPB importance and comprehensiveness in the modern high-performance computing landscape. In [12], the authors evaluated the performance of their CUDA/C porting of the FT and MG kernels, comparing the results with the original implementations. The authors in [13] used the MPI-Fortran version of seven NPB to propose a dynamic frequency scaling for energy consumption reduction in distributed applications. The paper in [14] uses NPB for

automatic parallelization and vectorization techniques included in the LLVM compiler as well as compares the performance of different compilers in a large set of applications including some kernels presented in the NPB suite. In contrast, we are porting to C++ the five NPB kernels and implementing the parallelism for multi-cores with three different parallel programming frameworks.

Authors from [15] proposed a tool to predict the best speed-up situation and thus reducing the cost of performance scalability evaluation. They utilized an unofficial C version of the NPB from the Omni Compiler Project [16] implemented over NPB 2.3 to evaluate their tool. That NPB version was implemented with a huge parallel region to reuse threads already spawned. The authors mentioned that it was a poor choice to evaluate their tool. So, they modified the NPB-C to attend to their needs. In the following research works from the same authors [17], [18], they continue using the same NPB-C obtained from the earlier work, which was never fully tested and evaluated the correctness of its computation. Concerning the work from [19], they characterized the performance of an OpenCL implementation over the NPB for a heterogeneous parallel platform that consists of general-purpose CPUs and a GPU. Thus, a version in C was needed of the NPB. No details were given about the NPB porting to C and its validation. Differently, from these studies, we are porting the NPB kernels based on the original Fortran code as well as fully testing and evaluating our implementations. Consequently, other researchers may use the sequential version and implement parallelism by using other parallel programming interfaces.

III. An Overview of NPB Kernels

In this section, we briefly describe each one of the five basic computational kernels to discuss our C++ porting and parallelization further.

A. Embarrassingly Parallel (EP)

The EP kernel generates a large number of Gaussian random pairs according to a specific scheme. Its intensive computation is concentrated in the main loop, which independently processes each group of random number pairs. Considering that some iterations of the loop have more numbers to generate than others, each iteration calculates its offset to balance the load. Then, uniform pseudo-random numbers are enumerated for each group to compute the Gaussian deviation by the acceptance-rejection method. After this, its computed solution is verified using previous results already validated in this kernel. This considers the number of Gaussian pairs generated and the independent total sum of X and Y Gaussian deviations [2], [5]. In sequence, we present the EP program steps:

1) Generates floating point values r_j in the interval (0, 1)
2) Set $x_j = 2r_{2j-1} - 1$ and $y_j = 2r_{2j} - 1$
3) Test if $t_j = x_j^2 + y_j^2 \leq 1$
4) If false, reject the current pair. If true: $k = k+1$, where $X_k = x_j\sqrt{\frac{(-2logt_j)}{t_j}}$ and $Y_k = y_j\sqrt{\frac{(-2logt_j)}{t_j}}$
5) X_k and Y_k are independent Gaussian deviates with mean zero and variance one
6) Q_l for $0 \leq l \leq 9$ are the pairs (X_k, Y_k) counting that belong to the square annulus $l \leq max(\lfloor |X_k|, |Y_k| \rfloor) \leq l+1$

B. Conjugate Gradient (CG)

The CG kernel computes an approximation with tolerance 10^{-10} of the smallest eigenvalue of a large, sparse, and unstructured matrix, exploiting the conjugate gradient method. Its intensive computation is indeed the conjugate gradient method that is represented by a partition submatrix multiply. The core computation remains in nested loops, which characterizes this kernel as a fine-grained one [2], [5], [20]. The following steps provide an overall idea of this kernel code and its operation sequence:

1) Generation of a sparse matrix $x[1, 1, ..., 1]$ with a random pattern of nonzeros
2) Utilization of the inverse power method which solves the system $Az = x$ and returns $\|r\|$ through the conjugate gradient method (described in 3). Then, it calculates $\zeta = \lambda + \frac{1}{xz}$, where λ is the shift for different problem sizes, and performs $x = \frac{z}{\|z\|}$
3) CG method, where Az represents the solution z to the linear system of equations $Az = x$:
   $z = 0, r = x, \rho = r^T r, p = r$
   do i = 1 to max
   $q = Ap$
   $delta = \frac{\rho}{p^T q}$
   $z = z + \delta p$
   $\rho_0 = \rho$
   $r = r - \delta q$
   $\rho = r^T r$
   $\beta = \frac{\rho}{\rho_0}$
   $p = r + \beta p$
   enddo
   $\|r\| = \|x - Az\|$

C. Integer Sort (IS)

IS performs an integer sort among a sparse set of numbers, which can be compared with particle-in-cell applications. By default, the sorting method is based on the bucket sorting approach. Accordingly, the number of keys for each bucket is determined, and the total count is distributed among each bucket. When completed, each bucket receives sorted numbers and points to the final accumulated sizes. Finally, the keys within each bucket are sorted, and a partial test is performed to verify the results [2] [20]. The steps are as follows:

1) Generation of the initial sequence of keys uniformly distributed in memory
2) Load all N keys into the memory system through the appropriate memory mapping
3) Computation of the sorting operation:
   do i = 1 to max
   a) Modify the sequence of keys:
      $K_i = i$ and $K_{i+max} = (B_{max} - i)$
   b) Computes each key rank
   c) For every iteration, performs the partial verification
   enddo
4) Execution of the full verification to evaluate the sorting operation.

D. MultiGrid (MG)

MG utilizes the multigrid method to compute a 3D Poisson equation. It also represents a V-Cycle algorithm with a residual performing n times to obtain an approximate solution to the discrete Poisson problem. Its routine consists in restricting the residual from fine to coarse grain to give an approximate solution by calculating the coarsest grid. Consequently, it extends the solution from a coarse grid to a fine grid. For each level, a residual is computed, and a smoothing operation is applied. After that, the final and more intensive computation perform the last k level, and a final residual is calculated [2], [5], [21]. The MG program steps are listed below:

1) Generation of a 3D matrix $v = 0$ in exception of n points that receive $v = \pm 1$
2) Each iteration evaluates the residual in $r = v - Au$ and applies a correction with $u = u + M^k r$
3) M^k represents the V_cycle multigrid operator (described in 4)
4) $z_k = M^k r_k$ where:
   a) if $k \leq 1$ then $z_1 = Sr_1$ (apply smoother) where S is the smoother operator
   b) else
   $r_{k-1} = Pr_k$
   $z_{k-1} = M^{k-1} r_{k-1}$
   $z_k = Qz_{k-1}$
   $r_k = r_k - Az_k$
   $z_k = z_k + Sr_k$

Here, A denotes the trilinear finite element discretization of the Laplacian ∇^2 normalized. Also, P is the trilinear projection operator of finite element theory and represents the half value of the operator Q.

E. Fourier Transform (FT)

This kernel performs a Fast Fourier Transform (FFT) of a 3D partial differential equation using spectra method. The core computation of FT is the inverse `cfftz` method, performed $niter$ times for each dimension. Also, a copy is made from the 3D matrix to a 1D matrix in each dimension. Then, the Fast Fourier Transform is computed, and the result is copied back. In the beginning, a function (`evolve`) performs the exponent factors for the inverse method. Finally, in the end, a checksum is computed to validate the result [2], [21]. The operations sequence is described as follows:

1) Generation of $2n_1n_2n_3$ random floating points and the $U_{j,k,l}$ 3D matrix is filled with those data
2) Computation of the forward 3D discrete Fourier transform (DFT) of U: $V = DFT(U)$
3) Computation $W_{j,k,l} = e^{-4\Pi^2 10^{-6}(\vec{j}^2+\vec{k}^2+\vec{l}^2)} V_{j,k,l}$, where $\vec{j}$ is j when $0 \leq j < \frac{n1}{2}$ and $j - n1$ when $\frac{n1}{2} \leq j < n1$. This repeats for k and l to $n2$ and $n3$
4) Calculation $X = inv(W)$, where inv represents the inverse 3D DFT
5) Computation of the checksum $\sum_{j=0}^{1023} X$

IV. C++ Porting and Parallelization

This section is devoted to present our C++ porting of the NPB kernels and the parallelization using C++ parallel programming frameworks.

A. C++ Porting

To implement parallelism in NPB with C++ parallel programming frameworks, we must port the original Kernels' Fortran code to C++. Our goal was to avoid significant differences when porting the code. For instance, maintain the same algorithmic structure, preserve the function definitions with the same procedure computation, and follow the same steps and order of the operations described before.

We could refer numerical algorithms to these benchmark kernels since they are composed by basic math operators, such as sums, subtractions, multiplications, divisions, powers, and roots. However, we needed to apply some modifications to avoid changing the original programming logic when porting the code from Fortran to C++. This is due the fact that each language has its own syntax to express the logic of the program (*e.g.*, the power operation is described as $\star\star$ in Fortran which is replaced by `pow()` in C++, and `mod()` in Fortran is represented by the `%` in C++).

Although we did much work for porting the original code, few differences remained in four out of the five kernels. The challenging kernel was FT. We restructured almost the entire code because Fortran has a primitive variable type to represent complex numbers and C++ has not. In our C++ version, we did not use the STL (Standard Template Library) to represent complex numbers. We preferred to implement a `struct` with two double variables to support the real and the imaginary part of a complex number. In this code porting, we also had to rewrite all functions that operate with the complex matrices. This includes operations such as addition, matrix initialization with sines and cosines, the Stockham Fast Fourier Transform, and others similar situations like this ones.

B. Parallelization

This section is about the parallel implementation of the five kernels. The instruments of our study include two important parallel programming frameworks. FastFlow is from research, and Intel Thread Building Blocks (TBB) is from the industry. Our OpenMP version is based on the original parallel Fortran code that is OpenMP too. TBB and FastFlow were designed with C++ STL to support parallel programmers with a high-level library interface.

NPB kernels are classified as data-parallel computations. The expected parallel patterns to be applied are `Map`, `Reduce`, and `MapReduce` operations. For expressing the parallelism with OpenMP, FastFlow, and TBB, we used the abstracted `Map` and `Reduce` interface, which has its way to express parallelism in each framework. For the kernels with a sequence of parallel regions inside a loop, we managed to reuse threads already spawned by the system in all the frameworks. Also, TBB and FastFlow can provide these patterns through lambda function.

1) EP: This kernel is composed by the main loop with minor communications between operations. Thus, we applied a `Map` pattern by using the parallel `for` loop of each framework. Following the original Fortran parallel version, we keep the static scheduler for OpenMP and FastFlow. In contrast, TBB uses the work-stealing scheduling and has a non-deterministic thread creation, always performing as a dynamic scheduler.

The parallel region starts right before the Gaussian deviates computation. Inside this region, the `MapReduce` parallel loop is annotated, which will reduce two variables (`sx` and `sy`). These variables contain the independent total sum of Gaussian deviations. A different `Reduce` must be performed with lock mechanisms to concatenate all partial count values from the computation of the Gaussian deviates. This was necessary since OpenMP only accepts variables as reduction parameters. Listing 1 exemplifies our OpenMP version. The parallel loop is depicted in line 2. OpenMP's reduction directive reduces the partial computation in lines 7 and 8 declared in the line 1. Also, line 10 shows where the critical directive is annotated which represents the step 6 described in Section III-A.

```
1  #pragma omp for reduction(+:sx,sy)
2  for (k = 1; k <= np; k++) {
3   /*abstracted code*/
4   for ( i = 0; i <= NK; i++) {
5    /*abstracted code*/
6    qq[l] += 1.0;
7    sx = sx + t3;
8    sy = sy + t4;
9   }
10  #pragma omp critical
11  for (i = 0; i <= NQ - 1; i++) q[i] += qq[i];
12 }
```

Listing 1. Representation of the parallelism in EP with OpenMP.

TBB's version extends the same OpenMP's approach to express the parallelism. The only difference is that our TBB version does not use the reduction as OpenMP does. We made use of the structure already implemented to reduce the total sum of Gaussian deviations of `sx`, and `sy` in addition to the count of the Gaussian deviates. In this implementation, we used the `tbb::mutex` mechanism provided by the TBB library to safely perform the `Reduce` operation.

In the FastFlow version, we also implemented the `MapReduce` pattern. However, differently, we used the `parallel_for_thid` method from the FastFlow library that provides the thread identifier number. Such low-level access allowed us to avoid the critical section and implement the reduction pattern as observed in Listing 2. This code represents the same piece of code previously presented for OpenMP in Listing 1. We selected this specific part of code from FastFlow to represent the contrast of the nonuse of lock mechanisms as in OpenMP and TBB. Lines 5, 6 and 7 represents the partial computation. The reduction inside the loop is depicted in line 10. We represent through `SCHED` the static scheduling in which we divided the maximum iteration size by the number of threads plus one.

```
1  pf.parallel_for_thid(1, np+1, 1, SCHED, [&](int k,
       int id) {
2   /*abstracted code*/
3   for (i = 0; i <= NK; i++) {
4    /*abstracted code*/
5    qq[id][l]._qq += 1.0;
6    sxx[id]._qq = sxx[id]._qq + t3;
7    syy[id]._qq = syy[id]._qq + t4;
8   }
9  });
10 for(i=0; i<num_workers; i++) {
11  sx += sxx[i]._qq;
12  sy += syy[i]._qq;
13  for(int j=0; j<=NQ-1; j++) q[j] += qq[i][j]._qq;
14 }
```

Listing 2. Representation of the parallelism in EP with FastFlow.

We faced the false sharing problem when expressing parallelism with FastFlow. To solve it, we implemented a `struct` to complete the default cache line size. In addition to that, we added to each position of the default queue, a blank character with size equal to the remaining proportion to fill the 64 bytes, which represents the default cache line size. This problem was avoided in OpenMP and TBB because each framework creates its thread private variables. In general, each framework tries to deal with low-level parallel programming issues to prevent the user from these details. Additionally, except for OpenMP, it is important to emphasize that the reduction could be implemented by the default `MapReduce` abstraction in TBB and FastFlow. However, it requires an implementation of all partial variables into a single `struct` or `class`. This is because their `Reduce` routine is implemented to manipulate one parameter at the time. Thus, the entire kernel would require modifications in regions that use these variables.

2) CG: Concerning EP, CG has a larger amount of code and more regions to look for parallelism. In the original code, this kernel contains several OpenMP `pragma` annotations during the code. We manage to maintain all `pragma` annotations for our C++ version. Thus, every loop inside the conjugate gradient method is a `map` pattern. Additionally, this kernel has two more intensive computations after the conjugate gradient method, which are represented by a `Map` and a `MapReduce` operation.

When parallelizing with FastFlow and TBB, we reduced the parallelism to three single regions containing the most intensive computation. These regions are inside the conjugate gradient method, in which the most intensive is represented by the $q = Ap$ operation that was described in Section III-B (a submatrix multiplication). The other two parallel regions represent the residual norm, which computes $||r|| = ||x - Az||$. Listing 3 depicts two out of the three intensive computations. The first loop inside line 1 and 6 represents the submatrix multiplication and the second loop inside 7 and 10 represents the part of the residual norm.

The submatrix multiplication is represented by the loop of the line 1 which performs the matrix iteration. In this case, dependencies are not the issue since the computation performs a local sum and saves the result for each index position into the vector `w[]` of the line 5. Therefore, we managed to use the `Map` pattern. For the other loop of the line 7, again a sum is performed. In this case, if executed in parallel, each thread will contain a partial result of the sum operation. Thus, we implemented a `MapReduce` pattern to perform this operation.

```
1  for (j = 1; j <= lastrow-firstrow+1; j++) {
2   sum = 0.0;
3   for (k = rowstr[j]; k < rowstr[j+1]; k++)
4    sum = sum + a[k]*p[colidx[k]];
5   w[j] = sum;
6  }
7  for (j = 1; j <= lastcol-firstcol+1; j++) {
8    d = x[j] - r[j];
9    sum = sum + d*d;
10 }
```

Listing 3. CG's submatrix (1 to 6) and residual norm (7 to 10).

For OpenMP and TBB, we utilized both abstracted patterns of each framework to introduce the parallelism. In FastFlow, we expressed the `MapReduce` parallelism not by its ab-

stracted form. We used its `parallel_for_thid` routine to access each thread's identifier and perform a custom `Reduce`.

3) IS: Once the original kernel code was already in C, minor modifications were needed. We observed that in its original implementation, the parallel version contains additional code concerning the serial. This was required to guarantee the communication between buckets, which were used as auxiliary structures to perform the sorting operation. In OpenMP, they made use of the thread identifier number to divide the workload statically. Additionally, we included a significant parallel region to retain the sorting operation's intensive computation for OpenMP. This way each thread has its data to pre-compute and to communicate. Thus, the intensive computation can safely be performed in parallel.

In Listing 4, we present a piece of code representing the computation described in item 3 from Section III-C, which performs the sorting operation. As can be observed, the `pragma` annotation in line 1 delimits a region of intensive computation that performs in parallel. Consequently, the subsequent loop in line 4 is the communication represented by a sequential computation, where each thread executes this routine once. This code between lines 4 and 10 is additional to the sequential IS kernel version and represents each thread sharing the results computed in line 3.

```
1  #pragma omp for schedule(static)
2  for(i=0; i<NUM_KEYS; i++)
3    work_buff[key_array[i] >> shift]++;
4  for(i=1; i< NUM_BUCKETS; i++ ) {
5    bucket_ptrs[i] = bucket_ptrs[i-1];
6    for(k=0; k< omp_get_thread_num(); k++ )
7      bucket_ptrs[i] += bucket_size[k][i];
8    for(k=omp_get_thread_num();k<omp_get_num_threads
       (); k++ )
9      bucket_ptrs[i] += bucket_size[k][i-1];
10 }
```

Listing 4. Representation of the parallelism in IS with OpenMP.

We were able to extend this strategy with the use of the `parallel_for_thid` routine in the FastFlow version. We had to introduce intermediate structures to perform the buckets communication and allocate a unique `Map` pattern for each intensive computation. In Listing 5 is depicted the code that follows the one presented in Listing 4. This piece of code is already implemented in FastFlow. The loop depicted in line 1 is where the parallelism is implemented for the intensive computation of line 3. Additionally, the next loop (line 5) is equivalent to the one described in the previous Listing 4 between lines 4 and 10 used for data synchronization. Note that the loop in line 5 represents a `Map` pattern, iterating from `zero` to `num_workers-1`. This structure mimics the parallel region of the OpenMP library where the thread executes a routine once.

```
1  pf->parallel_for_thid(0, NUM_KEYS, 1, SCHED, [&](
       int i, int id) {
2    INT_TYPE k = key_array[i];
3    key_buff2[bucket_ptrs2[id][k >> shift]++] = k;
4  });
5  pf->parallel_for(0,num_workers-1,1,[&](int myid){
6    for(INT_TYPE i=0; i< NUM_BUCKETS; i++ ){
7      for(INT_TYPE k=myid+1; k< num_workers; k++ )
8        bucket_ptrs2[myid][i] += bucket_size[k][i];
9    }
10 });
```

Listing 5. Representation of the parallelism in IS with FastFlow.

In TBB, we could not implement this kernel following the original strategy because thread manipulation is not accessible to the application programmer. Also, the threads are created in a non-deterministic way due to its work-stealing scheduling, which makes it even more difficult to reproduce in TBB.

4) MG: This intensive kernel computation remains inside the multigrid V-cycle routine. This is replication to the V-cycle multigrid operator described in Section III-D. Therefore, each V-cycle routine contains an internal loop in which k levels are performed, as depicted in Listing 6. The V-cycle iteration loop described in the line 1 would iterate the levels from the initial parameter until `lt-1`. Note that the last iteration is disengaged from the loop and is performed separately at the end (line 6). This iteration represents the most intensive computations, more than all the others together.

```
1  for (k = lb+1; k <= lt-1; k++) {
2    j = k-1;
3    zero3(); interp(); resid(); psinv();
4  }
5  j = lt - 1;
6  k = lt;
7  interp(); resid(); psinv();
```

Listing 6. Representation of the V-cycle multigrid routine.

The most of the OpenMP annotations were introduced inside the routines described in Listing 6, which contains several `pragma omp for` annotations. The `interp` function adds the trilinear interpolation of the correction, the `resid` computes the residual and the `psinv` applies the smoother. Each of those functions contains a part of the intensive computation, and to each of them, we express the parallelism independently. This segregation is because each operation needs a communication at the end (data updates). The reuse of threads already spawned plays an important role to achieve good performance in this kernel. In TBB and FastFlow, we also added the simple `parallel_for` lambda function inside these routines to express the parallelism.

5) FT: This kernel was modified to meet the complex data type defined in Fortran. Such modifications also changed the way to express parallelism in some functions with respect to the original version. All the parallel regions remain in the same scope, but we had to substitute some `MapReduce` by `Maps` with auxiliary structures to enable parallelism. In Listing 7, we selected the function which performs the FFT in one dimension. In this code, a copy of the 3D matrix is made to auxiliary structures, and at the end, it is copied back to the matrix. Inside this region, it performs the FFT method in the single dimension of data copied. As can be noted, we managed to represent the complex data type in C++ by implementing a `struct` with real (lines 5 and 10) and imaginary number (lines 6 and 11). In this example, the nested loop depicted in line 2 iterates with an offset of `fftblock`. We implemented the parallelism in this loop to try a better load balance. However, we obtained worst results than when paralyzing the external loop in line 1.

```
1  for (k = 0; k < d[2]; k++) {
2    for (jj = 0; jj <= d[1]-fftblock; jj+=fftblock)
       {
```

```
    for (j = 0; j < fftblock; j++)
     for (i = 0; i < d[0]; i++)
      y0[i][j].real = x[k][j+jj][i].real;
      y0[i][j].imag = x[k][j+jj][i].imag;
    cfftz (is, logd[0], d[0], y0, y1);
    for (j = 0; j < fftblock; j++)
     for (i = 0; i < d[0]; i++)
      xout[k][j+jj][i].real = y0[i][j].real;
      xout[k][j+jj][i].imag = y0[i][j].imag;
```

Listing 7. FFT method in a single dimension.

We expressed the parallelism similarly for all three parallel versions using the `Map` pattern. In the FT kernel, for all three dimensions of the matrix, it performs the FFT method described in the item 3 from Section III-E. After each iteration, a checksum is performed to partially evaluate the correctness of the result. For the OpenMP version, the checksum operation uses the critical directive to bypass OpenMP's reduction syntax. That is because OpenMP reduction directive does not support a `struct` data type as an entry argument.

For FastFlow and TBB, excluding the checksum operation that demands a `Reduce`, we applied the `Map` pattern to add parallelism for the most intensive computations, represented by the FFT method and some additional independent computations. Since the FFT method was decomposed into three independent operations performing one dimension at the time, we parallelized each dimension separately. Finally, we implemented the reduction in TBB (using `parallel_reduce`) and FastFlow (using `parallel_for_thid`) without introducing lock synchronizations.

V. Experiments

The experiments were executed in a machine equipped with 24GB of RAM and two processors Intel(R) Xeon(R) CPU E5-2620 v3 2.40GHz, with 6 cores each and support to hyper-threading, totaling 24 threads. Its operating system was Ubuntu Server 64 bits with kernel 4.4.0-59-generic. Moreover, we used GCC 5.4.0 with `-O3` compiler flag and the libraries: Thread Building Blocks (4.4 20151115) and FastFlow (r13).

In our tests, we used NPB's B class as parameters to compile the benchmarks[1]. This means that the kernels will assume: (1) 2^{30} random-number pairs for EP; (2) a grid with size of 512x256x256 and 20 iteration for FT; (3) IS will sort 2^{25} number of keys with maximum value of 2^{21}; (4) a grid with size of 256x256x256 and 20 iterations for MG; (5) and CG will have 75000 number of rows with 13 values non-zero and 75 iterations to perform with an eigenvalue shift of 60. Also, we ran each benchmark from 1 to the maximum degree of parallelism in the target machine. The execution was repeated 10 times for each degree of parallelism. With that, the obtained results represent the average execution time with properly standard deviations plotted in the graphs through error-bars.

A. Sequential Versions

Once the original kernels in Fortran were ported to C++, we measured the sequential version execution times to evaluate the efficiency of our code porting. Figure 1 presents the comparison between the C++ porting and the Fortran legacy code. Note that we achieved very similar results concerning the Fortran code, with less than 1% difference for all kernels. We can conclude that our C++ porting is similar to the original Fortran.

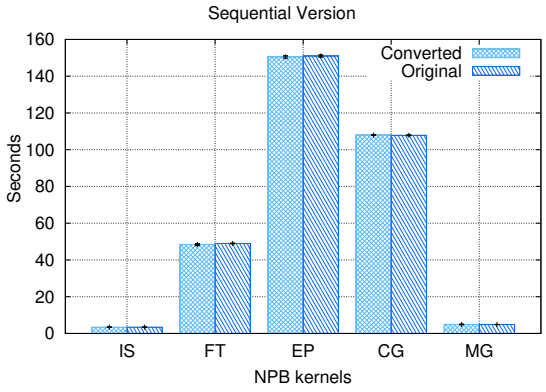

Fig. 1. Results of the porting to C++.

B. Parallel Versions

In the parallel versions, we used the serial C++ code to express the parallelism. The performance evaluation and comparison were made through the execution times plotted in the next graphs. The X-axis presents the degree of parallelism and Y-axis presents the time in seconds. Although we tried several ways to express the parallelism in each one of the frameworks and kernels, we will discuss the versions with the best performance. We considered the original parallel version with OpenMP (label `Original`), the C++ version with OpenMP (label `OMP`), the C++ version with FastFlow (`FF`), and the C++ version with TBB (label `TBB`). Also, we plotted the standard deviations through error bars.

Figure 2 shows the results of the parallelization for EP. These results reveled that the performance and scalability are similar in our OpenMP implementation with respect to the `Original` version. In FastFlow, the results were very similar too. However, TBB outperforms all other versions. We attribute this performance gain to the TBB work-stealing scheduler, which provides a better load balancing. We tested

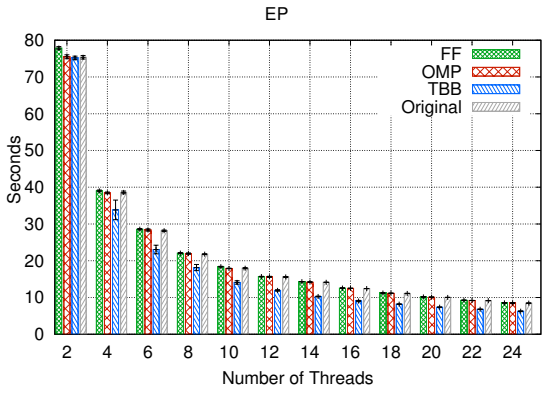

Fig. 2. EP with C++ parallel programming frameworks.

[1]The source codes for the serial and parallel versions are available at https://github.com/dalvangriebler/NPB-CPP

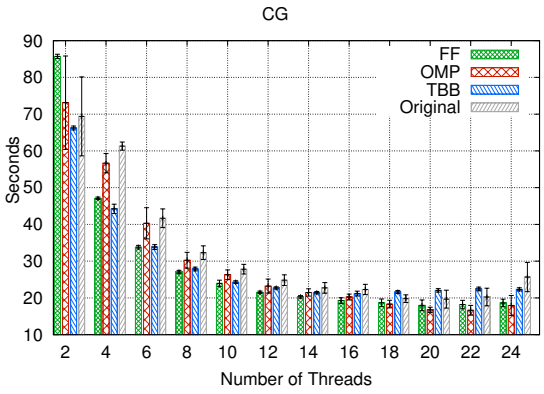

Fig. 3. CG with C++ parallel programming frameworks.

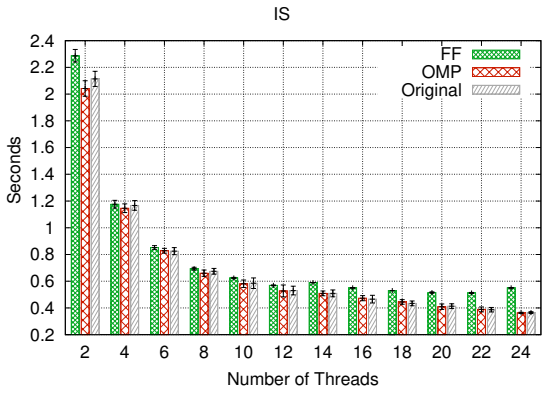

Fig. 4. IS with C++ parallel programming frameworks.

another scheduler in OpenMP and FastFlow, but no improvement was observed.

The CG kernel presented performance contrasts in Figure 3. There were also high standard deviations in the OpenMP versions (`Original` and `OMP`), which tends to be particular of the runtime. Note that the C++ parallel version achieved a slightly better performance than compared with the original Fortran parallelization. Except for the degree of parallelism 2, TBB and FastFlow had similar performance results. This was expected because both of them followed the same parallelism strategy although runtimes are working differently. Moreover, our C++ version of OpenMP achieved the best performance when using hyper-threading resources. This is related to the load balancing. Noted that OpenMP reaches better performance when there are more threads.

Concerning the IS kernel, results are shown in Figure 4. TBB does not appear in the graph because of the issues reported previously (Section IV-B3). There were small performance differences between the `OMP` and `Original` versions as it was for the serial version. On the other hand, FastFlow had performance degradation in the most of the degree of parallelism tested. That is because it has overhead when creating/waking-up threads frequently. This becomes evident when there is a small workload such as in IS (about milliseconds). Also, note that FastFlow results become worst when reaching hyperthreading. This indicates that its load balancing for a small workload is not as efficient as OpenMP.

The execution times of the parallel versions for MG are presented in Figure 5. The first impression is that this kernel has limited scalability. With more in-depth investigations, we discovered that the compiler could vectorize the most of the codes. In a certain point, the granularity becomes too small that the program stops to scale. This occurs even with the original version. However, it presented fewer overheads when increasing the degree of parallelism. FastFlow was able to achieve a similar performance concerning the original version, and also better than OpenMP and TBB. The poorest performance of TBB is related to the combination of its dynamic scheduler and computation small granularity.

Finally, FT results can be view at Figure 6. OpenMP in Fortran can take advantage of the complex number data type.

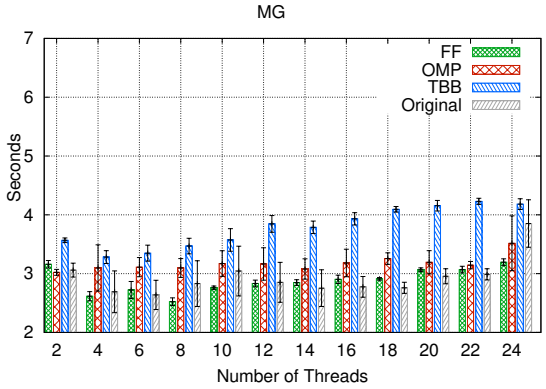

Fig. 5. MG with C++ parallel programming frameworks.

In C++, this is perceptibly worst when using OpenMP. That is because we must to introduce a critical directive to avoid race conditions. FastFlow achieved a similar performance with respect to the original version up to 12 threads. However, with more threads, it loses against the original version due to the load unbalancing that was the same problem for TBB and OpenMP.

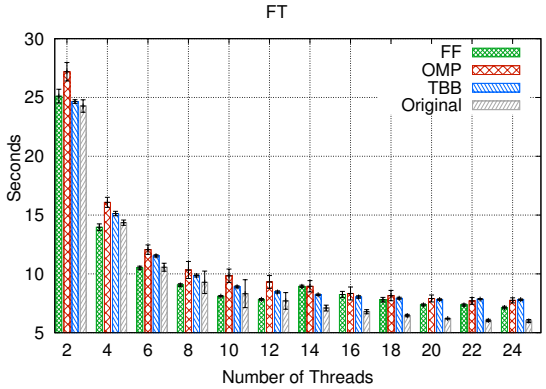

Fig. 6. FT with C++ parallel programming frameworks.

C. Final Remarks

In this paper, we were able to provide an efficient C++ version of the NAS benchmark kernels with a similar Fortran performance. In addition to OpenMP, we were able to express parallelism with C++ parallel programming frameworks such as TBB and FastFlow. Table I summarizes our performance results. By porting the kernel to C++, we identified how well the frameworks are optimized to exploit parallelism in the shared memory architecture. For EP, these results revealed that TBB's best speed-up is able to outperform (35 percent) the best speed-up of the original parallel Fortran code. In CG, OpenMP and FastFlow's best speed-ups also achieved better performance (18 and 9 percent respectively) than the original parallel Fortran code. As expected in IS, the C++ version of OpenMP's best speed-up has similar results concerning the C version of OpenMP's best speed-up. Finally, the C++ parallel versions for FT had worst (between 15 and 25 percent) performance than the original parallel Fortran code. In MG, FastFlow achieved the best speed-up although not performing very good.

TABLE I
A SUMMARY OF THE RESULTS.

Version	Metrics	FF	OMP	TBB	Original
EP	Threads	24	24	24	24
	Speed-up	17.53	17.59	23.84	17.67
	Time (s)	8.53	8.50	6.27	8.46
CG	Threads	20	22	16	20
	Speed-up	5.98	6.46	5.08	5.47
	Time (s)	17.97	16.64	21.17	19.66
IS	Threads	23	24	-	24
	Speed-up	6.70	9.47	-	9.42
	Time (s)	0.51	0.36	-	0.36
MG	Threads	8	5	3	5
	Speed-up	1.94	1.69	1.50	1.87
	Time (s)	2.52	2.90	3.25	2.61
FT	Threads	24	22	24	23
	Speed-up	6.74	6.25	6.16	8.10
	Time (s)	7.13	7.70	7.80	5.93

VI. Conclusions

This paper presented a performance evaluation of the NAS benchmark kernels using C++ parallel programming frameworks. We ported the legacy Fortran code to C++, providing comparable performance. Our parallelization in the C++ converted code achieved better performance than the parallel Fortran code from the original version for the EP, CG, IS, and MG kernels. These results highlight the main contribution of our work, where we allow other frameworks to be tested with a well-established benchmark (NPB) in the modern computer architectures. As future work, we plan to run these kernels in different machine architectures, parallelize these kernels for accelerators, include in the parallelization other C++ frameworks, port to C++ the NPB's pseudo-applications, and test the performance with other workloads.

Acknowledgment

The authors would like to thank the partial financial support of CAPES, FAPERGS, PUCRS and of the EU H2020 project RePhrase (EC-RIA, ICT-2014-1). We also thank Massimo Torquati for the fruitful discussion about FastFlow.

References

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

[2] 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, and S. Weeratunga, "The NAS Parallel Benchmarks," NASA Ames Research Center, Moffett Field, CA - USA, Tech. Rep., March 1994.

[3] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," *SIGARCH Comput. Archit. News*, vol. 23, no. 2, pp. 24–36, 1995.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54.

[5] H. Jin, M. Frumkin, and J. Yan, "The OpenMP Implementation of NAS Parallel Benchmarks and its Performance," NASA Ames Research Center, Moffett Field, CA - USA, Tech. Rep., October 1999.

[6] D. De Sensi, T. De Matteis, M. Torquati, G. Mencagli, and M. Danelutto, "Bringing Parallel Patterns Out of the Corner: The P3ARSEC Benchmark Suite," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 4, pp. 33:1–33:26, Oct. 2017.

[7] M. Danelutto, T. De Matteis, D. De Sensi, G. Mencagli, and M. Torquati, "P3arsec: Towards parallel patterns benchmarking," in *Proceedings of the Symposium on Applied Computing*, ser. SAC '17. New York, NY, USA: ACM, 2017, pp. 1582–1589. [Online]. Available: http://doi.acm.org/10.1145/3019612.3019745

[8] J. Reinders, *Intel Threading Building Blocks*. USA: O'Reilly, 2007.

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

[10] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "FastFlow: High-Level and Efficient Streaming on Multi-core," in *Programming Multi-core and Many-core Computing Systems*, ser. PDC, vol. 1. Wiley, March 2014, p. 14.

[11] D. Griebler, M. Danelutto, M. Torquati, and L. G. Fernandes, "SPar: A DSL for High-Level and Productive Stream Parallelism," *Parallel Processing Letters*, vol. 27, no. 01, p. 20, March 2017.

[12] L. Wang, M. Huang, V. K. Narayana, and T. El-Ghazawi, "Scaling scientific applications on clusters of hybrid multicore/gpu nodes," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, ser. CF '11. New York, USA: ACM, 2011, pp. 6:1–6:10.

[13] J. C. Charr, R. Couturier, A. Fanfakh, and A. Giersch, "Dynamic frequency scaling for energy consumption reduction in synchronous distributed applications," in *IEEE International Symposium on Parallel and Distributed Processing with Applications*, Aug 2014, pp. 225–230.

[14] A. Y. Drozdov, S. V. Novikov, V. E. Vladislavlev, E. L. Kochetkov, and P. V. Il'in, "Program auto parallelizer and vectorizer implemented on the basis of the universal translation library and llvm technology," *Programming and Computer Software*, vol. 40, no. 3, pp. 128–138, 2014.

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

[16] "Omni compiler project," Nov 2017. [Online]. Available: http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/

[17] M. Popov, C. Akel, W. Jalby, and P. de Oliveira Castro, "Piecewise holistic autotuning of compiler and runtime parameters," in *Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing*, Aug 2016, pp. 238–250.

[18] M. Popov, C. Akel, Y. Chatelain, W. Jalby, and P. de Oliveira Castro, "Piecewise holistic autotuning of parallel programs with cere," *Concurrency and Computation: Practice and Experience*, vol. 29, Aug 2017.

[19] S. Seo, G. Jo, and J. Lee, "Performance characterization of the nas parallel benchmarks in opencl," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, Nov 2011, pp. 137–148.

[20] S. Saini and D. H. Bailey, "NAS Parallel Benchmark (Version 1.0) Results 11-96," NASA Ames Research Center, Moffett Field, CA - USA, Tech. Rep., November 1996.

[21] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmakrs 2.0," NASA Ames Research Center, Moffett Field, CA - USA, Tech. Rep., December 1995.