BLonD++: Performance Analysis and Optimizations for Enabling Complex, Accurate and Fast Beam Dynamics Studies

Konstantinos Iliakis[*]
CERN
Geneva, Switzerland
konstantinos.iliakis@cern.ch

Helga Timko
CERN
Geneva, Switzerland
helga.timko@cern.ch

Sotirios Xydis
National Technical University of Athens
Athens, Greece
sxydis@microlab.ntua.gr

Dimitrios Soudris
National Technical University of Athens
Athens, Greece
dsoudris@microlab.ntua.gr

ABSTRACT

This paper focuses on the performance analysis and optimization for enabling efficient implementations of next generation beam dynamics simulations. Nowadays large worldwide research centers, e.g. CERN, Fermilab etc. are continuously investing in resources and infrastructures for progressing knowledge in the fields of particle physics, thus requiring careful studies and planing for upcoming upgrades of the synchrotrons and the design of future machines. Consequently, there is an emerging need for simulations that incorporate a collection of complex physics phenomena, produce extremely accurate predictions while keeping the computing resources and run-time to a minimum. A variety of simulator suites have been developed, however, they have been reported to lack in simulation speed, features and ease-of-use. In this paper we introduce the Beam Longitudinal Dynamics (BLonD) simulator suite from a computer engineering perspective. We analyze its performance to understand its current bottlenecks and enhance it further in an attempt to make complex, accurate and fast beam dynamics simulations possible. We show that through careful and targeted analysis and code tuning, the proposed BLonD++ implementation delivers significant gains in terms of performance, i.e. up-to 23× single-core speedup and scalability, thus enabling the deployment of even more complex simulations than the current state-of-art.

KEYWORDS

Multi-core Processors, Scientific Simulators, Beam Dynamics

ACM Reference Format:
Konstantinos Iliakis, Helga Timko, Sotirios Xydis, and Dimitrios Soudris. 2018. BLonD++: Performance Analysis and Optimizations for Enabling Complex, Accurate and Fast Beam Dynamics Studies. In *Proceedings of International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XVIII) (SAMOS'18).* ACM, New York, NY, USA, 8 pages. https://doi.org/

[*]Also with National Technical University of Athens.

1 INTRODUCTION

The Large Hadron Collider (LHC) is the world's largest and most powerful particle accelerator. It started up on September 2008 and remains the latest addition to CERN's accelerator complex. LHC is used to collide high-energy particles in order to study the fundamental laws of particle physics. A series of particle accelerators, mainly circular machines called synchrotrons or injectors, are used to accelerate these particles to increasingly higher energies. The injectors of the LHC in order are: the Proton Synchrotron Booster (PSB), the Proton Synchrotron (PS) and the Super Proton Synchrotron (SPS). As these machines are working at very different energy ranges, the beam motion is dominated by different physics phenomena in each machine.

The collection of charged particles in the accelerators is called a beam and interacts with the conducting material of the machine in which it is circulating. Beam dynamics is the field of physics that describes the beam motion in particle accelerators. The beam consists of several bunches, and each of these bunches is a compact collection of charged particles, kept together by a large external radio frequency (RF) voltage.

With the upgrade projects of the CERN synchrotrons [9, 12] and studies of future machines [2], there is a growing need for precision simulations that can combine for a given study all the relevant physics effects with the machine-specific features. At the same time, the simulation suite has to be general enough to cover a large range of synchrotrons, from low- to high-energy regimes, accelerating protons, electrons or ions.

To fulfill these critical requirements, the Beam Longitudinal Dynamics simulation suite (*BLonD*) [1] has been developed at CERN since 2014. It focuses on the longitudinal plane of the beam and tracks the energy and time coordinates. Written in Python and C++, this tracking code has a modular structure that allows the user to include different physics models depending on what the study requires. Furthermore the *BLonD* simulator is an open-source project and supports essentially every platform that disposes a Python interpreter and a C compiler.

By now, through extensive application for beam studies in- and outside of CERN, the *BLonD* suite has been thoroughly benchmarked [23]. The benchmarks performed range from comparisons with theory and measurements, to other particle tracking codes and increased the trust in *BLonD* and its predictions. The outcome of the simulation studies is continuously guiding the baseline choices for machine upgrades and future machines.

For instance, for the upcoming upgrade of the SPS [20], scientists rely heavily on the results of *BLonD* simulations to give predictions on what can be achieved in the future. In many cases the machines are being pushed beyond their designed limits. The upgraded systems are designed with minimal margins in order to be as cost-efficient as possible. Thus, simulations need to be very accurate, despite the complexity of the machines. Also, whenever new phenomena in operational machines are discovered it is crucial to have a tool that can reproduce and explain observations.

To cope with the overwhelming simulation complexity and prediction accuracy needed in the field of longitudinal beam dynamics, this paper focuses on optimizing the run-time performance of the current state-of-art longitudinal beam dynamics simulator, the *BLonD* suite. We focus on four realistic simulation scenarios, each concerning a different particle accelerator, and use them as testcases for our performance analysis. As a first approach to performance optimization, the computationally most intensive parts of the code are ported to C++. Then, the Top-Down method [26] is applied on the testcases to provide an in-depth micro-architectural insight of *BLonD*. Based on the analysis outcome, we identify a series of bottlenecks and proceed to mitigate them through compiler tuning, use of high performance scientific libraries and other software optimization techniques. In addition, we employ OpenMP [8] to parallelize the compute intensive code regions. *BLonD*++, the proposed implementation, combines an efficient C++ computational core with a higher-level control-flow code written in Python. Finally, we evaluate the single-core run-time performance of *BLonD*++ as well its scalability in a multi-core Intel Haswell [13] server platform. The proposed implementation demonstrates an up to 23× single-core run-time speedup. By dramatically reducing the duration of a week-long simulation to below 9 hours, *BLonD*++ has enabled several beam dynamics studies that were previously unfeasible due to run-time, memory and CPU limitations.

The paper is organized as follows: Section 2 reviews prior art on beam longitudinal dynamics simulators. Section 3 provides an overview of the structure of the *BLonD* suite. In Section 4, a detailed description of the methodology we followed to tackle the current limitations of *BLonD* is presented. The experimental evaluation of *BLonD*++ in terms of single-core performance and scalability takes place in Section 5. Section 6 concludes this paper.

2 RELATED WORK

At CERN, scientists needed a highly customizable tool-set for simulations to drive a large range of longitudinal beam dynamics studies. The physics features that required to be modeled were not implemented in any other code, thus the development of the *BLonD* simulator suite started back in 2014.

Prior to *BLonD*, ESME [17] was widely used in the area of longitudinal beam dynamics. Developed at Fermilab since 1984, the code is written in Fortran and is compatible with few Unix-based operating systems like Solaris. Some common features between ESME and *BLonD* have been benchmarked in terms of correctness and the results show total agreement [23]. Due to lack of support, maintenance and development of new features since 2011, the scientific community is adopting *BLonD* more and more.

Another alternative is the Py-orbit [21] code. Py-orbit is a "Particle-In-Cell" (PIC) code and is a mix of python and C++. As it is a 6D tracker, it can also model transverse beam dynamics, but consequently, the computational complexity of a py-orbit simulation is significantly heavier than a *BLonD* simulation. Good agreement between the two codes has been reported in the literature [10, 23]. Being a general-purpose code, py-orbit lacks many of the specific longitudinal features available in *BLonD*. Similarly to py-orbit, Elegant [7] is a 6D tracking code too. It was developed since 2000 at the Argonne National Laboratory. Elegant is a mature code in terms of performance optimizations as it has been parallelized with MPI [25] and features a GPU accelerated version [16].

The *BLonD* suite differs from the aforementioned codes in numerous ways. The python front-end makes *BLonD* easy-to-use and particularly attractive to new users. The modular structure allows rapid prototyping of new features that extend its capabilities. Contrary to py-orbit and elegant, *BLonD* specializes in the longitudinal plane and as a consequence, it contains more detailed physics models and is computationally less heavy, resulting in shorter simulation times. Finally, *BLonD* has been tested successfully on a wide range of real-world simulation scenarios.

3 BACKGROUND

3.1 The *BLonD* suite architecture

Figure 1 summarizes a simple particle accelerator model containing the three main components of *BLonD*: the ring, the RF section(s) and the beam. Following the object-oriented design, these components are represented by different classes in the code. Their interactions, the most essential ones being shown in Fig. 1, are usually modeled by a method. In some cases, where heavy parametrization is needed, an interaction is separated into a class.

The ring itself is described by the properties that contain information amongst others about the acceleration and the machine impedance which can interact with the beam current and produce an induced voltage.

A beam can be composed of several bunches made up of charged particles. In reality, a bunch can contain trillions of particles, however, in simulations macro-particles are used which represent many real particles in order to reduce the memory footprint. The user is responsible for determining the amount of macro-particles required to describe a certain physical phenomenon with sufficient resolution. The computational complexity of most operations in a *BLonD* simulation scales linearly with the number of simulated macro-particles, which typically ranges from 500 thousand to 100s of millions.

RF sections are placed in fixed locations along the ring, and this is where the particles are accelerated; the required energy being fed by large RF amplifiers. The acceleration, meaning the increase of the particle energy, is described by either the continuous `kick()` or the linearly interpolated `LIkick()` operation. The latter is a

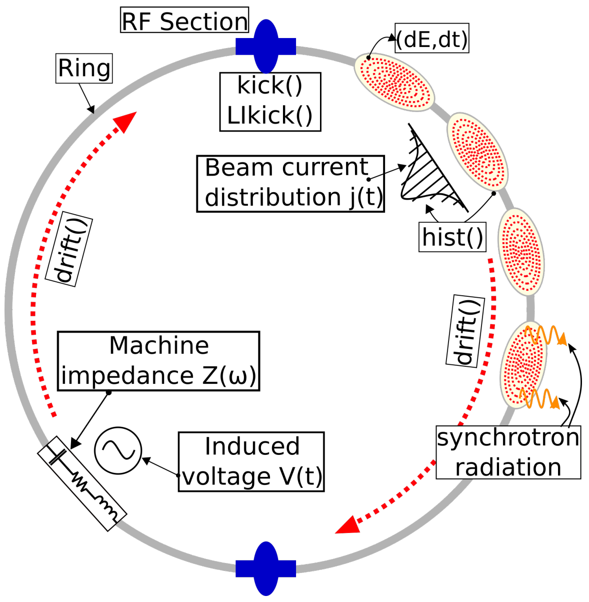

Figure 1: Simplistic synchrotron model. The ring, the beam and the RF sections are the main components. Their interactions are simulated with *BLonD* on a turn-by-turn basis.

simplification that replaces an expensive trigonometric function calculation with a pre-calculated look-up table. The number of RF stations used depends on the modeled case and ranges typically from one to a dozen. The beam motion from one RF station to another is modeled by the `drift()` operation. The rigorous equations of motion modeled in *BLonD* can be found in the work of H. Timko et al. [22]. A full cycle of the beam corresponds to a single simulation iteration. The number of iterations required for a given study can rage from a few thousand to a few million.

Along the machine there can be various sources of impedance, for instance the beam pipe itself is made of conducting material and interacts with the beam current distribution, that is the particle charge histogram along the time coordinate. The latter is calculated by the `hist()` method. The beam-impedance interaction is most efficiently modeled in the frequency domain, for which forward and backward fast Fourier transformations are needed.

In synchrotrons, magnets are used to bend the beam trajectory, and this bending causes the charged particles to radiate off part of their energy. This phenomenon is called synchrotron radiation (`SR()`) and might or might not be negligible depending on the studied case.

BLonD is a modular and flexible library. A *BLonD* simulation scenario is an assembly of components which in turn can be composed of smaller sub-parts. The user, knowing which physics effects are essential for a given study, initializes the relevant components. Then, the user wires the components to form a pipeline of physics transformations that will be computed on a turn-by-turn basis. Some optional features of *BLonD* include a complete tool-set for data analysis, storage and plotting.

Table 1: Basic configuration and run-time of the selected testcases in the python-only version.

Testcase	Turns	Particles	RF Stations	Run-time
PSB	500K	2M	1	~3 days
SPS	100K	72M	1	~2 weeks
LHC	14M	600K	1	~10 days
FCC	100K	1M	2	~16 hours

3.2 Realistic testcases

The runtime performance of a simulation highly depends on the physics to be modeled, or in other words, on what modules the user choses to combine. This is why, for the performance analysis of the *BLonD* suite, we chose to analyze four realistic cases that cover a wide range of CERN applications. Each case considers a different particle accelerator. These are: PSB, SPS, LHC and the Future Circular Collider (FCC). The latter is a 100 km collider that is presently under study.

PSB. The PSB is the lowest energy synchrotron, dominated by so-called space-charge effects that require a large amount of simulated macro-particles. It requires the use of the continuous version of `kick()` to avoid numerical noise.

SPS. The interaction with the machine impedance requires to model around 100 bunches with at least 1M macro-particles each resulting in a memory footprint of more than 1.2 GB. As a result this simulation is the most time consuming per iteration.

LHC. The simulation of the acceleration ramp of the LHC requires 14M iterations with relatively fewer macro-particles. As part of the study, beam losses are calculated by the `stats()` module.

FCC. The particularity of this case is the significant energy losses due to synchrotron radiation requiring several sub-iterations over one revolution period. To model the synchrotron radiation effect a pseudo-random number generation (PRNG) function with a huge repetition period is needed.

Table 1 summarizes the basic configuration of each testcase. Each study requires typically scanning a large parameter space. As a consequence, tens to hundreds or even thousands of runs are needed for a complete beam dynamics study.

4 PERFORMANCE ANALYSIS AND OPTIMIZATION

In this section we provide a straightforward guideline to performance optimization adopted in the development of *BLonD*++. The stages of the guideline are applicable to essentially any large-scale scientific application. What needs to be tailored to the specific case is the realization of each stage. The suggested methodology is depicted in Fig. 2.

4.1 Moving to a C++ runtime

BLonD started originally as a pure python code, as python combines rapid prototyping and development, ease-of-use, and object oriented design principles. Moreover, it can be easily extended by third-party libraries that contain a rich collection of mathematical tools [14, 24]. Figure 3 summarizes the run-time breakdown

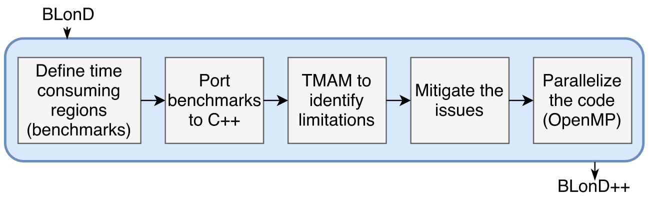
BLonD
Define time consuming regions (benchmarks)
Port benchmarks to C++
TMAM to identify limitations
Mitigate the issues
Parallelize the code (OpenMP)
BLonD++

Figure 2: Performance optimization methodology.

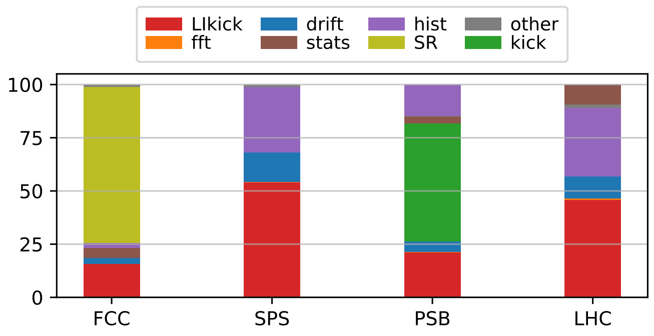

Figure 3: Run-time breakdown of the four target testcases with the initial, python-only *BLonD* version. The seven tagged methods are responsible for 99% of the run-time on average.

of the four targeted testcases. The seven tagged methods, namely `LIkick()`, `drift()`, `hist()`, `fft()`, `stats()`, `SR()` and `kick()` aggregate 99% of the simulation time. Thus, for the rest of the paper these seven methods will be referred to as benchmarks and the overall performance of *BLonD* will be improved by optimizing each benchmark individually or collectively whenever possible.

As a first approach to reduce the run-time, the selected benchmarks were ported to C++; a programming language well-suited for performance-critical applications. To interface the existing Python code with the C++ extensions, the C++ sources are compiled into a shared library which is exposed to Python via the `ctypes` module. This hybrid implementation combines the best of both worlds; the usability of Python in the front-end and the efficiency of C++ in the compute intensive back-end.

A noticeable speedup was achieved by porting the computationally intensive core to C++ and is reported in Fig. 4. The first bar of every group, shows the overall speedup of the testcase specified in the x-axis and the rest bars of every group show the speedup of each individual benchmark in that testcase. The run-time has been reduced by 3.3× up-to 12.5× or 7.5× on average. The `SR()` method of the FCC testcase used the Boost library [19] for the PRNG as it was noticed at this early stage that the STD library PRNGs were lacking in performance. The `fft()` benchmark has not been optimized with respect to the python-only version as at that moment, `fft()` was allocating a very slight percentage of the run-time.

4.2 Introduction to TMAM

In this sub-section, we provide a brief overview of the Top-down Micro-architecture Analysis Method (TMAM) [26]. TMAM is used

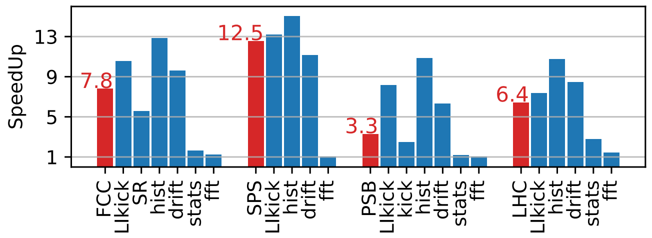

Figure 4: Per testcase and per-benchmark speedup of the first *BLonD*++ revision compared to the initial python-only version.

to identify performance limitations of software in modern, out-of-order (O3) processors. TMAM divides the total number of available processor pipeline slots into four categories:

Bad Speculation denotes slots wasted due to all aspects of incorrect speculations like mis-predicted branches.

Retiring denotes slots utilized by "useful operations". Ideally, all slots should be attributed here. A high retiring fraction does not necessary mean that there is no room for improvement.

Front-End Bound denotes stalled slots because the pipeline's front-end under-supplies the back-end. The front-end is the portion of the pipeline responsible for fetching the next instruction from the ICache and decoding it into micro-operations to be executed by the back-end.

Back-End Bound denotes stalled slots due to lack of resources to accept new operations. It is further divided into: **Memory bound** which reflects execution stalls due to the cache and memory subsystems, and **Core bound** which reflects either pressure on the execution units or lack of ILP.

The advantages of TMAM compared to other approaches are that TMAM is generic enough to be applied to any modern O3 processor, it induces a low-cost time overhead, and it offers clear insights on performance bottlenecks.

4.3 Identification & mitigation of performance limitations

In Section 4.1, a compelling speedup was reported by porting the computation core of the *BLonD* code to C++. To go even further, we need to identify the performance limitations of the new code. In this section, the TMAM analysis is applied to the targeted testcases in order to better understand and eventually tackle the micro-architectural bottlenecks of the *BLonD* suite.

To reproduce the TMAM break down described in Section 4.2, a collection of approximately 90 hardware counters is needed. To facilitate the automation of the collection process, the command-line interface of the Intel VTune Amplifier [3] was used. The Instrumentation and Tracing Technology (ITT) API [5] was utilized to localize the event collection around the regions of interest as well as enable a more fine-grained, per-benchmark event grouping. Finally, after the collection and grouping of the events was completed, the formulas suggested by TMAM were used to breakdown the total available processor pipeline slots into the following categories: front-end bound (FEB), bad speculation (BS), retiring (RET), core bound (CB) and memory bound (MB).

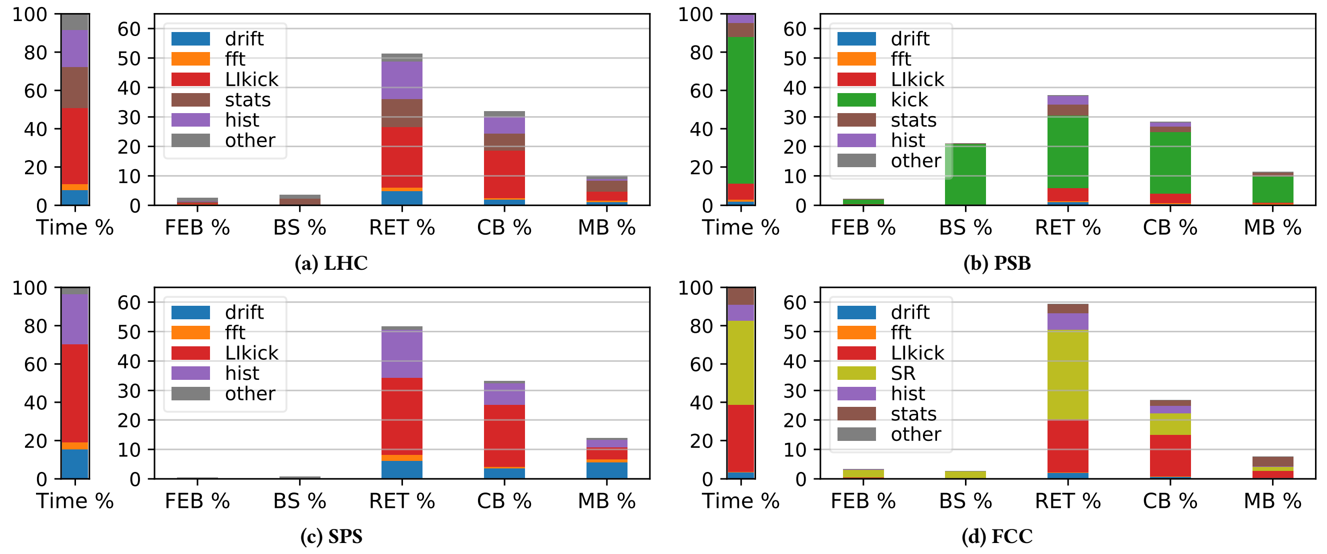

Figure 5: Breakdown of available processor pipeline slots according to TMAM for the four targeted testcases.

Figure 5 shows this breakdown for each of the targeted testcases displaying the percentage of the total available pipeline slots dedicated to each of the above mentioned categories on a per-benchmark granularity. The bars on the left each plot show the time contribution of the considered benchmarks to the run-time. Based on Fig. 5, the following five performance inefficiencies were identified.

4.3.1 Core-bounded `LIkick()`. The `LIkick()` benchmark is in overall responsible for a big portion of wasted cycles due to core related stalls. Core-related stalls are usually caused by sequences of dependent instructions or unbalanced use of the execution units that leads to instruction serialization and ILP deterioration. Furthermore, `LIkick()` contributes significantly to the retiring part. As mentioned in Sec. 4.2, a high retiring percentage does not necessarily mean that there is no space for improvement. In particular, vectorization is a technique that lets more operations to be executed by a single instruction, thus decreasing the retiring percentage and speeding up the execution at the same time.

The inefficiency spotted in `LIkick()` was tackled in two phases. The `LIkick()` function computes the energy transferred to each macro-particle, every time the beam passes through an acceleration cavity. At first, it was noticed that a portion of the main computation of `LIkick()` was independent of the particle index and was determined only by the particle distribution bin to which the particle belonged to. As a result, two helper arrays of a size equal to the number of bins ($\sim 10^3$) were pre-calculated and then used as look-up tables in the main loop ($\sim 10^6 iterations$) saving expensive computations. Furthermore, the main loop was unrolled to enable partial vectorization.

4.3.2 Memory-bounded `drift()`. The second identified issue is related to the `drift()` benchmark. Figure 6 shows that `drift()` suffers from frequent memory stalls. To mitigate this pathogenic behavior, we noticed that the calculation of `drift()` and `LIkick()` can be interleaved. By doing so, the memory loads are reduced

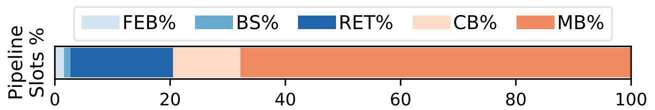

Figure 6: TMAM breakdown for the `drift()` benchmark. 68% of total pipeline slots is wasted due to memory-related stalls. The input size was 1M particles.

roughly by 50% and therefore the pressure to the memory subsystem is reduced.

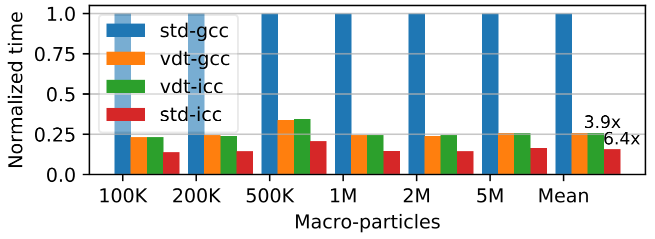

Figure 7: Comparison of STD and VDT libraries with the gcc and icc compilers in `kick()`. The STD-icc configuration is on average 6.4× faster than the STD-gcc configuration.

4.3.3 Inefficient `kick()` *implementation.* The third issue concerns the `kick()` benchmark that dominates the run-time of the PSB testcase. The most time consuming part of `kick()` is the calculation of the `sin()` function. In Fig. 7, the performance of `kick()` is evaluated with the C++ Standard Library (STD) [15] and the VDT Library [18] compiled with the gcc and icc compilers. The values on the y-axis are normalized to the `STD-gcc` configuration. The fastest configuration appears to be the use of the STD library compiled with the gcc compiler which is on-average 6.4× faster than the `STD-gcc` configuration.

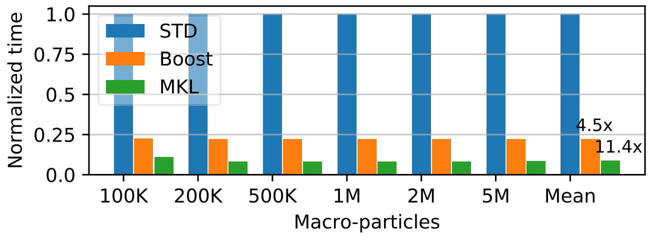

Figure 8: Benchmarking the STD, Boost and MKL libraries for the PRNG methods in the `SR()` function. The MKL PRNG method is on average 11.4× faster than the STD counterpart.

4.3.4 Inefficient `SR()` *implementation.* The `SR()` benchmark is responsible for the fourth bottleneck. In the FCC testcase, `SR()` dominates in the RET, FEB, BS categories and the run-time. The most time-consuming task of the `SR()` benchmark is the pseudo-random number generation (PRNG). In Fig. 8, the performance of three different PRNG libraries is evaluated: STD [15], Boost [19] and Intel MKL [4]. The latter is the most efficient and outperforms the STD library by 11.4× on average across a range of input sizes.

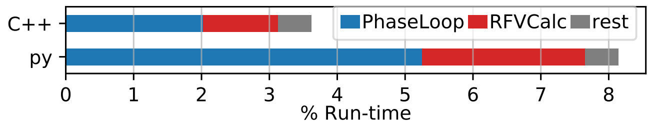

Figure 9: Breakdown of the "other" part of the LHC testcase before and after porting `PhaseLoop()` and `RFVCalc()` to C++.

4.3.5 Large contribution of "other" in LHC testcase. The final underlined limitation appears in the LHC testcase. The "other" part, which represents the code that does not belong to any of the considered benchmarks, allocates 8% of the run-time. While this might seem as a minor issue, it is crucial to reduce the contribution of the serial parts as they greatly affect the overall scalability of the code. With detailed profiling, we discovered the two most significant methods of the "other" part: `RFVCalc()` and `PhaseLoop()`. By porting them to C++ the contribution of the "other" part to the overall run-time dropped to 3.5%. Figure 9 summarizes the run-time breakdown of the "other" part in the LHC testcase, before and after the porting to C++.

Tackling the issues mentioned above with the suggested techniques lead to the next generation beam longitudinal dynamics simulator suite, *BLonD*++. The evaluation of the single-core performance of *BLonD*++ is given in Section 5.2.

4.4 Code parallelization

To anticipate forthcoming computational challenges in the field of beam dynamics, the seven considered benchmarks were parallelized.

The framework used to express parallelism is the OpenMP [8]. Simplicity, maturity, compiler support, and scalable performance are some of the assets that made OpenMP the most popular shared-memory parallelization framework. In general, most benchmarks were parallelized using the parallel loop `pragmas`. Some benchmarks, like `hist()` were not inherently parallel as they need to update shared data structures. In this case, to avoid atomic operations or other means of synchronization, each thread computes a private histogram and in the end the private histograms are reduced to a global one. For the `fft()` benchmark, the multi-threaded version of the FFTW library [11] was used. We performed a deep evaluation and analysis of *BLonD*++ scalability, reported in Section 5.3.

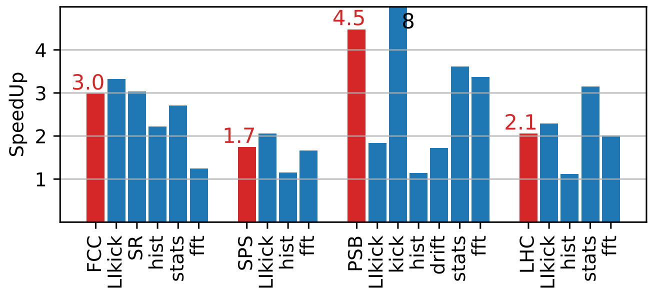

Figure 10: Single core execution time evaluation of *BLonD*++ after mitigating the identified issues with TMAM.

5 EXPERIMENTAL EVALUATION

5.1 Experimental setup

Table 2: Hardware Set-up

Model	Intel® Xeon® Haswell E5-2683v3 @ 2.00GHz
CPU	2 nodes with 14 cores per node 2-way Hyper-Threading total 28 cores/ 56 threads
Cache	32KB L1I and L1D per core 256KB L2 per core 35MB L3 shared per node
Memory	64 GB
Operating System	CentOS Linux 7.4, kernel 3.10
Compiler	gcc 5.3 & icc 18.0, compile flags: -O3 -ffast-math -mtune=native

The proposed *BLonD*++ library is evaluated experimentally on a NUMA, multi-core server platform. Table 2 summarizes the hardware set-up. The Intel Turbo Boost technology and the hyper-threading feature were disabled for stable and reproducible measurements. The standard deviation of the reported results is ≈ 1%.

5.2 *BLonD*++ single-core performance

Figure 10 presents the speedup gained by mitigating the identified bottlenecks described in section 4.2.

The `Likick()`, now includes `drift()` in all testcases except PSB, has been improved by a factor of 2× to 3× in terms of run-time. This is mainly the result of saving computations by utilizing the look-up tables described in section 4.3, reducing memory loads by

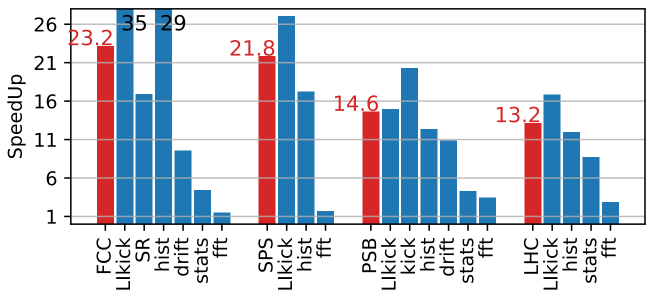

Figure 11: Single-core cumulative speedup of *BLonD++* over the initial *BLonD* version. *BLonD++* demonstrates a 18× speedup on average.

Table 3: Efficiency of *BLonD++* scalability.

Testcase	Serial%	Efficiency %		
		4	14	28 threads
FCC	0.85	95	80	73
PSB	3.73	95	85	84
SPS	0.35	93	47	44
LHC	9.40	93	80	81
Mean	3.58	94	73	71

overlapping `LIkick()` with `drift()` and finally employing auto-vectorization. The `kick()` benchmark that was dominating the PSB testcase is 8× faster in the fully optimized version. In the FCC testcase, the `SR()` has been improved by a factor of 3× due to the use of the random number generation functions from the MKL [4] library. Note that in the previous, un-optimized version, the Boost [19] library was used and not the C++ STD [15] library. In the `fft()` benchmark, the Scipy [14] FFTs have been replaced by the more efficient FFTW [11] library. The `fft()` bench it demonstrates a speedup of up to 3.3×.

In Fig. 11, the cumulative speedup of the final *BLonD*++ version against the initial python-only version is presented. *BLonD*++ achieved a 18× speedup in the four representative testcases on average. This means that a previously day-long, single-core simulation, can now be completed in 80 minutes while a week-long simulation needs only 9h to complete. Furthermore, this dramatic reduction in execution time has enabled the scientists using *BLonD* to simulate scenarios that combine more complex physics phenomena with finer resolution. For instance, in the SPS, modeling 144 bunches was crucial to get more accurate predictions for the upgrade of the machine, since bunches are coupled through intensity effects. This was not possible with the initial *BLonD* version.

5.3 Scalability analysis

The dramatic single-core speedup reported in the previous section is sufficient to enable studies of physics effects in deeper detail that were previously unfeasible due to run-time limitations. Nevertheless, as future challenges are anticipated, the considered benchmarks were parallelized to provide even greater speedups.

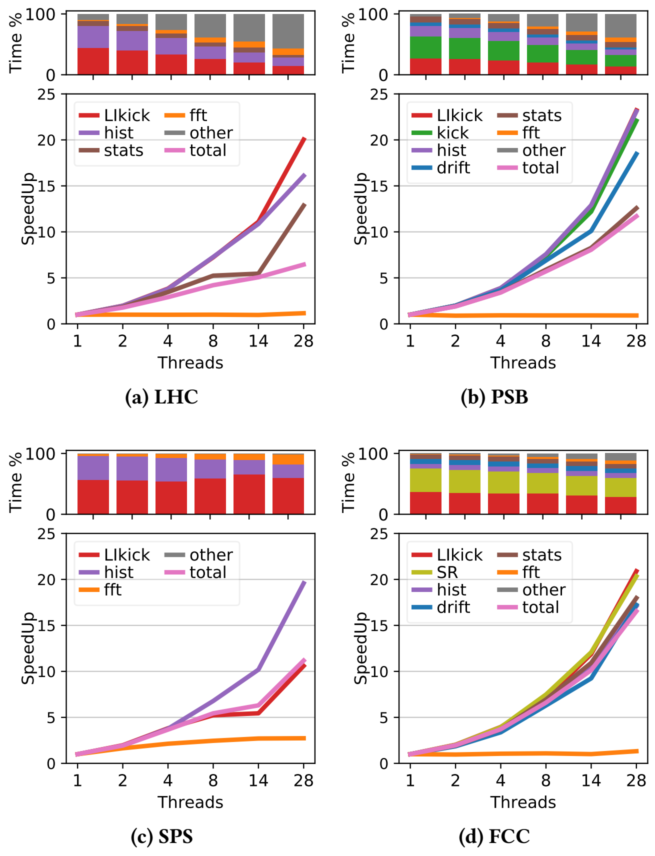

Figure 12: Scalability of the four realistic cases.

As mentioned in Section 4.3, OpenMP was used to express parallelism. The multi-threaded version of FFTW [11] was used in the `fft()` benchmark. FFTW enables multi-threading only after a certain threshold of input points. This is why the `fft()` benchmark shows scalable behavior only in the SPS testcase where the problem size is large enough. The FFTW library also defines the exact number of utilized threads depending on the input size.

Figure 12 summarizes the scalability analysis for the four selected testcases. The top sub-plot of each sub-figure shows how the contribution of each benchmark to the total execution time changes with the increase of the thread count. The contribution of the multi-threaded parts decreases with the increase of the threads and correspondingly, the non-parallelized parts become more significant as the thread count increases. This means that for a testcase to be scalable as a whole, all of its subparts need to be sufficiently scalable. The bottom sub-plot of each sub-figure shows the speedup of each benchmark as well as the speedup of the whole testcase, compared to the single threaded execution.

In general, most of the benchmarks demonstrate decent scalability. However, not all testcases scale well as a whole. In the LHC and PSB testcases the "other" part, dominates the run-time when the thread count increases. The "other" part needs to be further broken down into sub-parts, analyze them and explore new optimization opportunities. In the SPS testcase, `LIkick()` does not demonstrate a very scalable behavior. This is mainly due to the big input size which requires more than 1.2 GB of memory. As the thread count increases, the overall performance is limited by the memory bandwidth. This explains the speedup jump from 14 to 28 threads. The first 14 threads are allocated in the first node to avoid expensive inter-thread communication via the main memory. The second group of threads is scheduled in the second node, which unlocks an extra 35MB of L3 cache. As a result, the performance of this testcase scales ideally from 14 to 28 threads. Finally, the FCC testcase demonstrated the most scalable behavior among the testcases. Tackling the above mentioned issues is on-going work.

Table 3 shows how efficiently each testcase scales. The "Serial%" column shows what percentage of the run-time is allocated by serial code. The efficiency has been calculated as the ratio of the measured speedup to the theoretical speedup according to Amdahl's law [6]. All the reported values are percentages. We show the efficiency for four, 14 and 28 threads. Four threads is a typical number of cores that a desktop computer has, 14 is the number of cores in each node of the platform used for the experimental evaluation and 28 is the total number of available cores in the experimental platform. On average, *BLonD*++ achieved near-optimal, 94% scalability efficiency with four threads. This indicates that the multi-threaded benchmarks are indeed efficiently parallelized, successfully avoiding harmful effects of synchronization and load imbalance. Moreover *BLonD*++ achieved over 70% efficiency with 14 and 28 cores which is acceptable considering that 14-cores is the total number of cores per slot and 28-cores is the total number of cores in the system. Nonetheless, there is still room for improvement.

6 CONCLUSION

This paper introduced for the first time the *BLonD* simulation suite from a computer engineering point of view. After setting-up the essential background, an in-depth micro-architectural analysis of *BLonD* is provided. Based on the micro-architectural profile, the major limitations of *BLonD* were underlined. A series of techniques like vectorization, code overlapping to reduce memory loads, employment of look-up tables to save computations, use of high performance scientific libraries, and other compiler-related optimizations were suggested to mitigate the identified bottlenecks.

The implementation of these techniques led to *BLonD*++, a hybrid Python-C++ code that combines the ease-of-use of Python in the front-end with the efficiency of a compiled language like C++ in the back-end. The single-core performance of *BLonD*++ was evaluated in a multi-core server platform and demonstrated an up to 23× speedup in run-time. Furthermore, the scalability analysis of *BLonD*++ showed promising results.

In particular, a previously day-long simulation can now be completed in 80 minutes while a week-long simulation needs only 9h to complete, without even employing thread parallelism. This dramatic reduction in execution time has enabled the scientists using *BLonD* to simulate scenarios that combine more complex physics phenomena with finer resolution. These complex, accurate and fast simulations in the field of beam dynamics are essential to overcome the current limitations, plan the up-coming particle accelerators' upgrades and design future machines that will help science advance even further.

REFERENCES

[1] 2014. CERN Beam Longitudinal Dynamics code BLonD. (2014). https://blond.web.cern.ch/
[2] 2014. The Future Circular Collider study. *CERN Courier* 54, 3 (Apr 2014), 16–18. http://cds.cern.ch/record/2064538
[3] 2017. Intel® VTune™ Amplifier 2017. (2017). https://software.intel.com/en-us/intel-vtune-amplifier-xe
[4] 2018. Intel® Math Kernel Library. (2018). https://software.intel.com/en-us/mkl
[5] 2018. ITT API Open Source. (2018). https://software.intel.com/en-us/articles/intel-itt-api-open-source
[6] Gene M Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference.* ACM, 483–485.
[7] Michael Borland. 2000. *Elegant: A flexible SDDS-compliant code for accelerator simulation.* Technical Report. Argonne National Lab., IL (US).
[8] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.
[9] H Damerau, A Funken, R Garoby, S Gilardoni, B Goddard, K Hanke, A Lombardi, D Manglunki, M Meddahi, B Mikulec, G Rumolo, E Shaposhnikova, M Vretenar, and J Coupard. 2014. *LHC Injectors Upgrade, Technical Design Report, Vol. I: Protons.* Technical Report CERN-ACC-2014-0337. http://cds.cern.ch/record/1976692
[10] Vincenzo Forte, Danilo Quartullo, Alessandra Lombardi, and Elena Benedetto. 2015. Longitudinal injection schemes for the CERN PS Booster at 160 MeV including space charge effects. (2015).
[11] Matteo Frigo and Steven G Johnson. 1998. FFTW: An adaptive software architecture for the FFT. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, Vol. 3. IEEE, 1381–1384.
[12] Apollinari G., Bejar Alonso I., Bruning O., Fessia P., Lamont M., Rossi L., and Tavian L. 2017. *High-Luminosity Large Hadron Collider (HL-LHC): Technical Design Report V. 0.1.* CERN, Geneva. http://cds.cern.ch/record/2284929
[13] Per Hammarlund, Alberto J Martinez, Atiq A Bajwa, David L Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, et al. 2014. Haswell: The fourth-generation intel core processor. *IEEE Micro* 34, 2 (2014), 6–20.
[14] Eric Jones, Travis Oliphant, and Pearu Peterson. 2014. {SciPy}: open source scientific tools for {Python}. (2014).
[15] Nicolai M Josuttis. 2012. *The C++ standard library: a tutorial and reference.* Addison-Wesley.
[16] JR King, IV Pogorelov, KM Amyx, M Borland, and R Soliday. 2017. GPU acceleration and performance of the particle-beam-dynamics code Elegant. *arXiv preprint arXiv:1710.07350* (2017).
[17] JA MacLachlan. 1992. *Particle tracking in E {minus}{phi} space for synchrotron design and diagnosis.* Technical Report. Fermi National Accelerator Lab., Batavia, IL (United States).
[18] Danilo Piparo, Vincenzo Innocente, and Thomas Hauth. 2014. Speeding up hep experiment software with a library of fast and auto-vectorisable mathematical functions. In *Journal of Physics: Conference Series.*
[19] Boris Schäling. 2011. *The boost C++ libraries.* Boris Schäling.
[20] Elena Shaposhnikova, Joël Repond, Helga Timko, Theodoros Argyropoulos, Thomas Bohl, and Alexandre Lasheen. 2016. Identification and Reduction of the CERN SPS Impedance. (2016).
[21] Andrei Shishlo, Sarah Cousineau, Jeffrey Holmes, and Timofey Gorlov. 2015. The particle accelerator simulation code PyORBIT. *Procedia Computer Science* 51 (2015), 1272–1281.
[22] H. Timko, S. Albright, T. Argyropoulos, K. Iliakis, I. Karpov, A. Lasheen, D. Quartullo, J. Repond, M. Schwarz, and J. Esteban Müller. 2018. Beam Longitudinal Dynamics Simulation Suite BLonD. *Physical Review Accelerators and Beams (to be published)* (2018).
[23] Helga Timko, Danilo Quartullo, Alexandre Lasheen, and Juan Esteban Müller. 2016. Benchmarking the beam longitudinal dynamics code BLonD. (2016).
[24] Stéfan van der Walt, S Chris Colbert, and Gael Varoquaux. 2011. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering* 13, 2 (2011), 22–30.
[25] Yusong Wang and Michael Borland. 2006. PELEGANT: A parallel accelerator simulation code for electron generation and tracking. In *AIP Conference Proceedings*, Vol. 877. AIP, 241–247.
[26] Ahmad Yasin. 2014. A top-down method for performance analysis and counters architecture. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on.* IEEE, 35–44.