Characterization and Optimization of the Fitting of Quantum Correlation Functions

Pi-Yueh Chuang and Niteya Shah
Dept. of Computer Science
Virginia Tech
Blacksburg, Virginia, USA
{pychuang, niteya}@vt.edu

Patrick Barry and Ian Cloët
Physics Division
Argonne National Laboratory
Argonne, IL, USA
{barry, icloet}@anl.gov

Emil M. Constantinescu
Mathematics & Computer Science Division
Argonne National Laboratory
Argonne, IL, USA
emconsta@anl.gov

Nobuo Sato and Jian-Wei Qiu
Theory Center
Jefferson Laboratory
Newport News, VA, USA
{nsato, jqiu}@jlab.org

Wu-chun Feng
Dept. of Computer Science
Virginia Tech
Blacksburg, Virginia, USA
wfeng@vt.edu

***Abstract*—This case study presents a characterization and optimization of an application code for extracting parton distribution functions from high energy electron-proton scattering data. Profiling this application code reveals that the phase-space density computation accounts for 93% of the overall execution time for a single iteration on a single core. When executing multiple iterations in parallel on a multicore system, the application spends 78% of its overall execution time *idling* due to load imbalance.**

We address these issues by first transforming the application code from Python to C++ and then tackling the application load imbalance via a hybrid scheduling strategy that combines dynamic and static scheduling. These techniques result in a 62% reduction in CPU idle time and a 2.46× speedup in overall execution time per node. In addition, the typically enabled power-management mechanisms in supercomputers (e.g., AMD Turbo Core, Intel Turbo Boost, and RAPL) can significantly impact intra-node scalability when more than 50% of the CPU cores are used. This finding underscores the importance of understanding system interactions with power management, as they can adversely impact application performance, and highlights the necessity of intra-node scaling tests to identify performance degradation that inter-node scaling tests might otherwise overlook.

***Index Terms*—C++, Python, parallelization, profiling, characterization, optimization, performance, power management, scalability, systems, deep inelastic scattering, quantum physics.**

I. Introduction

Parton distribution functions (PDFs) are a particular class of quantum correlation functions (QCFs) defined within the theory of quantum chromodynamics (QCD) that describe the longitudinal momentum fractions of quarks and gluons (collectively known as partons) inside moving hadrons such as protons and neutrons. These functions play a key role in many aspects of particle physics in facilities such as Jefferson Lab, the Relativistic Heavy Ion Collider, the Large Hadron Collider, and will be important for studies of hadron structure at the upcoming Electron Ion Collider (EIC).

PDFs, however, are not calculable from first principles but can be reconstructed from high-energy particle scattering data. This task, known as QCD global analysis, has been carried out by several groups around the world [1]–[4] and remains an active area of research in QCD phenomenology. With the Jefferson Lab 12-GeV program and the future EIC, the field of hadronic physics is entering a new era of exploration of the 3D hadron structure via transverse momentum-dependent PDFs [5] and generalized parton distribution functions [6]. One of the challenges is that these quantities need to be reconstructed from multi-dimensional phase-space densities (or events), which in turn are reconstructed from experimentally measured particle momentum vectors resulting from collision events. The reconstruction of these phase-space densities, known as unfolding, is challenging due to the presence of detector effects and backgrounds, which ultimately impose irreducible systematic uncertainties.

Instead of reconstructing the phase-space density from events, an alternative approach is to carry out a simulation-based analysis. By comparing the phase-space samples from a simulation with those from the experiment, one can implement an inference strategy on QCFs and bypass the limitations imposed by the unfolding procedures. However, the numerical complexity of such simulation-based analysis requires an efficient numerical framework.

In this work, we consider a case study using theory codes from the Jefferson Lab angular momentum collaboration (JAM) [7] and present a performance and scalability characterization of the application code for extracting PDFs from simulated event-level data.

Our main contributions include the following:

- Characterizing and optimizing an application code for extracting parton distribution functions (PDFs) within the theory of quantum chromodynamics (QCD), resulting in a 62% reduction in CPU idle time and a 2.46× speedup in overall per-node execution time.

- Exposing the potentially adverse impact of power management on intra-node performance and scalability (e.g., a fully threaded CPU under full load will drop its CPU frequency to maintain CPU power consumption, but, in turn, impact per-thread performance).
- Highlighting the importance of analyzing and understanding single-node scaling behavior, as CPU frequency drops under full workload can be overlooked in coarse-grained node-based scaling studies.

The rest of the paper is organized as follows. §II provides the mathematical context to understand the nature and complexity of the problem. §III describes the application code and our characterization practices. §IV presents our performance and scalability results. Finally, §V summarizes our findings.

II. Background

Because the main focus of this work is a performance case study, we only briefly describe the mathematical nature of the underlying problem that our application code aims to solve. The mathematical components discussed here are then mapped to the components in the high-level flowchart in Fig. 1 and presented in §III. We only show the formulation of key components for brevity. Exact formulations for other functions can be found in [8], [9].

A. PDFs from Deep Inelastic Scattering

We consider the process of deep inelastic scattering (DIS), where a beam of electrons scatters off a beam of protons or neutrons. Using detectors around the interaction region, the momentum of the scattered electron can be recorded as data. The reaction can be schematically written in momentum space as follows:

$$e^-(l) + N(P) \to e^-(l') + X(W) \tag{1}$$

Here, l and P are the 4-momenta of the incoming electron and nucleon N (i.e., proton or neutron), respectively, and l' denotes the final state detected electron. X represents the unobserved remnants of the collision with total momentum W, resulting from the breaking of the proton and the transformation of its parts into other forms of matter. In practice, it is convenient to work with two sets of variables that allows one to relate PDFs with the underlying reaction. Specifically we use

$$Q^2 = -q^2 = -(l - l')^2, \qquad x = \frac{Q^2}{2P \cdot q} \tag{2}$$

to isolate the relevant degrees of freedom that characterize the reaction. Specifically, QCD factorization theorem provides a theoretical formulation for the phase-space density proportional to the so-called *differential cross section* given by

$$\frac{d\sigma^N}{dxdQ^2} = \sum_i \int_x^1 \frac{d\xi}{\xi} H_i\left(\frac{x}{\xi}, Q^2, \alpha_S(\mu^2), \mu^2/Q^2\right) f_{i/N}\left(\xi, \mu^2\right) \tag{3}$$

Here, N labels the reacting hadron and the sum over i runs over all possible partonic constituents, i.e., up, down, strange, charm, and bottom quarks and their corresponding anti-quarks and gluons. H_i is a quantity calculable in perturbative QCD (pQCD) in powers of the strong coupling α_S. $f_{i/N}$ is the PDF, which can be interpreted as the number density of finding a parton of type i inside a hadron N with a momentum fraction between ξ and $\xi + d\xi$. The scale dependence μ on the right-hand side (RHS) of Eq. (3) is optimized by choosing $\mu = Q$, which accounts for larger logarithmic corrections present in the calculation of H_i. This is achieved by solving a set of equations for the strong coupling and the PDFs which reads

$$\frac{d\alpha_S(\mu^2)}{d\ln\mu^2} = -\sum_{i=1} \beta_i \alpha_S^{i+2}(\mu^2)\ , \tag{4}$$

with the coefficients β_i calculable in pQCD. This equation can be solved numerically using empirically determined values for the strong coupling at a given scale. The scale dependence of the PDFs, on the other hand, is more involved as they obey a system of integro-differential equations known as DGLAP (Dokshitzer-Gribov-Lipatov-Altarelli-Parisi) evolution equations. These equations can be solved analytically in Mellin space [10] and their solutions have the form

$$\tilde{f}_{i/N}(n, \mu^2) = \sum_j U_{ij}(n, \mu^2, \mu_0^2) \tilde{f}_{j/N}(n, \mu_0^2) \tag{5}$$

The sum runs over all parton flavors, and $\tilde{f}_{j/N}$ are Mellin transforms of the PDFs, i.e., $\tilde{f}_{i/N}(n) = \int_0^1 d\xi \xi^{n-1} f_{i/N}(\xi)$. The evolution operator U_{ij} can be calculated in pQCD [10] and encodes the transformation of the PDFs from the input scale μ_0 to any other scale μ. We can then write the entire differential cross-section in Eq. (3) in Mellin space as

$$\begin{aligned}\Sigma^N(n, Q^2) &\equiv \int_0^1 x^{n-1} \frac{d\sigma^N}{dxdQ^2} \\ &= \sum_{i,j} \tilde{H}_i(n, Q^2, ...) U_{ij}(n, ...) \tilde{f}_{j/N}(n, \mu_0^2)\end{aligned} \tag{6}$$

using the Mellin transforms of H_i. The expression can be numerically inverted to the original space via

$$\frac{d\sigma^N}{dxdQ^2} = \frac{1}{\pi} \int_0^\infty dz\ \mathrm{Im}\left[e^{i\phi} x^{-n(z)} \Sigma^N(n(z), Q^2)\right] \tag{7}$$

using a complex contour parameterized as $n(z) = c + z\exp(i\phi)$ with constant values for c and ϕ chosen to optimize the convergence of the integral over z. The integral can be performed numerically using Gaussian quadrature using sub-intervals to increase precision.

The remaining task is then to model the input scale PDF in Eq. (6). For this, we use a standard parameterization by the JAM collaboration. Specifically, for each parton flavor, we use

$$\tilde{f}_{i/N}(n, \mu_0^2 | \theta_{0...4}^{i/N}) = \theta_0^{i/N} \frac{\tilde{T}(n, \theta_{1...4}^{i/N})}{\tilde{T}(a+1, \theta_{1...4}^{i/N})} \tag{8}$$

with the shape function given as linear combinations of beta functions [11]. θ's are the free parameters to be optimized for

the PDF inference task. We can now combine Eqs. (7) and (8) and obtain a final master formula

$$\frac{d\sigma^N}{dxdQ^2} \simeq \mathcal{T}^N(x,Q^2,\boldsymbol{\theta}^N) \equiv \frac{1}{\pi}\sum_{k,l} w_l J_l^k \mathrm{Im}\left[e^{i\phi}x^{-n_l^k} \sum_{i,j}\tilde{H}_i(n_l^k,Q^2,...)U_{ij}(n_l^k,...)\theta_0^{j/N}\frac{\tilde{T}(n_l^k,\theta_{1...4}^{j/N})}{\tilde{T}(a+1,\theta_{1...4}^{j/N})}\right] \quad (9)$$

The sub-intervals of the z integration ranges are labeled with k, while the index l labels the Gaussian quadrature points with weights w_l. The factor J_l^k is the Jacobian transformation to implement Gaussian quadrature within the limits of integration of the z-range sub-intervals. Notice that all the target dependence N is ultimately encoded in the PDF parameters, as required by theory.

The formulation based on QCD factorization is only accurate within a certain region of the physical phase space. For instance, Q^2 needs to be larger than typical hadronic masses and the invariant mass of the hadronic debris $W^2 = (q+P)^2$ needs to be large enough to avoid the so-called *resonance* region. We employ typical cuts employed in QCD phenomenology where $Q > Q_{\rm cut} = 1.28$ GeV and $W^2 > W^2_{\rm cut} = 10$ GeV^2. Also, the inference task requires us to include the normalization of the cross-section within the region of exploration including the kinematic cuts. We refer to this quantity as $\sigma_{\rm cut}$, which can be measured experimentally and computed using Eq. (9) as

$$\sigma^N_{\rm cut}(\boldsymbol{\theta}^N) = \int dxdQ^2\ \mathcal{T}^N(x,Q^2,\boldsymbol{\theta}^N)\ \Theta(Q^2_{\rm cut},W^2_{\rm cut}) \quad (10)$$

with a Heaviside step function Θ that implements phase-space cuts. In practice, reconstructing all quark flavors and anti-quark PDFs requires the consideration of additional reactions, which is beyond the scope of this work. We focus on reconstructing the up and down quark PDFs and the gluon, assuming phenomenologically known values for the strange quark PDFs. To achieve this, we consider proton and neutron beams, with the latter serving as a proxy for deuteron beams, as neutron beams are not available in practice. This allows the use of isospin relations that relate $f_{u/p} = f_{d/n}$, $f_{d/p} = f_{u/n}$, and similarly for anti-u and anti-d quarks, while the rest of the quarks and gluons are the same for both protons (p) and neutrons (n). These relations allow us to obtain $\boldsymbol{\theta}^n$ from $\boldsymbol{\theta}^p$, and thus, we only need to perform inference for the $\boldsymbol{\theta}^p$ parameters.

B. Event-Level Inference for $\boldsymbol{\theta}^p$

To carry out an inference for the proton PDF parameters $\boldsymbol{\theta}^p$ using the master formula, Eq. (9), with event samples from proton and neutron beams, we employ a standard unbinned maximum likelihood analysis and include the corresponding fiducial cross-sections. Specifically, the objective function to be optimized is as follows:

$$L(\boldsymbol{\theta}) = -\sum_{N=p,n}\omega_{1,N}\left[\frac{1}{M_N}\sum_{i=1}^{M_N}\ln\left(\frac{\mathcal{T}^N(x_{N,i},Q^2_{N,i},\boldsymbol{\theta}^N)}{\sigma^N_{\rm cut}(\boldsymbol{\theta}^N)}\right)\right] + \sum_N \omega_{2,N}\left|\frac{\sigma^N_{\rm cut}-\sigma^N_{\rm cut}(\boldsymbol{\theta}^N)}{\delta\sigma^N_{\rm cut}}\right| \quad (11)$$

where the phase-space samples $x_{N,i}$, $Q^2_{N,i}$, and cut cross-sections $\sigma^N_{\rm cut}$, $\delta\sigma^N_{\rm cut}$ are given as training samples from simulated DIS experiments on protons and neutrons. We use the limited-memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS) algorithm from SciPy to optimize the loss function in Eq. (11). The optimization is bounded, as the PDF parameters $\boldsymbol{\theta}$ need integrability conditions along with additional considerations to render the PDF realistic. (See [11] for more details.)

III. Approach

The first part of this section provides a brief description of the solution algorithms used in the original application code, which was written in Python+NumPy. The second part focuses on the practices that we apply to improve the performance of the application code.

A. Design of the Application Code

Fig. 1 illustrates the design and modularization of our QuantOm project application code [12], showing the implementation of finding the PDF parameters by minimizing Eq. (11). The PDF & DGLAP block and the $\mathsf{T}^{\mathsf{N}}(\dots)$ block, which form the Theory module, correspond to the calculations from Eqs. (3) to (9). The Scoring block represents the log terms in Eq. (11), while the Integrator block handles the numerical integration of Eq. (10). The optimizer used is the L-BFGS algorithm, as noted in §II.

Starting with the PDF Parameters block and proceeding through the Theory module, the workflow forms a feedback loop that continues until the PDF parameters stabilize. Once the loop terminates, the parameters from the last iteration are treated as the solution. It is important to note that the application does *not* execute the workflow in Fig. 1 just once. Each run involves numerous independent executions of the workflow, as described below.

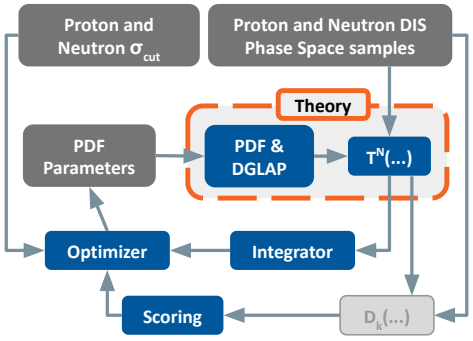


Fig. 1: A high-level overview of the components in the target application code. The process begins with pseudo-data — synthetic data generated from known parton distribution functions (PDFs) to mimic experimental data for solution verification. Proton and neutron deep inelastic scattering (DIS) phase-space samples are represented as sets of $[\,x\ Q^2\,]$. Additional synthetic data, proton and neutron $\sigma_{\rm cut}$, are generated using Eq. (10) with ground-truth PDFs. Note that the block $\mathsf{D}_{\mathsf{k}}(\dots)$, representing simulators of real-world detectors in experiments, is not used and thus not discussed in this work. PDF parameters are iteratively updated using the L-BFGS optimization algorithm until convergence to the desired tolerance.

Experimental events (pseudo-data in this work) represent a continuous probability distribution, requiring an infinite number of events to precisely recover the distribution function. A limited number of events results in some regions being under-represented, causing uncertainty in the fitted PDFs. To quantify this uncertainty, we apply a bootstrapping-style ensemble analysis, as illustrated in Fig. 2. The PDF Fitting block represents one execution of the workflow. The Bootstrap Sampler block draws a new set of events from the original dataset with replacement, matching the number of events in the original dataset. Each bootstrapped dataset is processed independently from the others. We denote the fitting of a bootstrapped dataset as one *bootstrap*.

The current implementation of the bootstrap distribution uses MPI. Although there is no communication between bootstraps, MPI allows finer control over the random number generators by binding rank information with the random seed, ensuring reproducibility of the results. The distribution of bootstraps uses static scheduling, meaning each rank's bootstraps are predetermined and fixed.

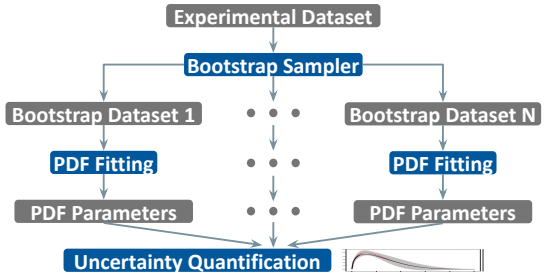


Fig. 2: A visual illustration of ensemble analysis via bootstrapping. The Bootstrap Sampler generates new datasets by drawing events, with replacement, from the original dataset. Each bootstrap dataset undergoes an independent fitting process. Once all bootstrap datasets are fitted, the collection of final PDF parameters can be used for further analysis to understand the uncertainty contributed by the limited experimental data.

B. Performance Characterization and Optimization

Starting with the original Python code from our quantum physicists on the QuantOm project, we use `cProfile` to profile and identify bottlenecks. To work from the same reference point with respect to the algorithm, we then port the Python code, along with these bottlenecks, to C++. After briefly characterizing the C++ implementation, we focus on its parallelization, followed by a deeper performance characterization to optimize its parallel performance. This staggered approach ensures the immediate release of stable application prototypes, including intermediate ones, to our end users.

In addition, the scalability of this and subsequent application codes will be crucial for deployment on exascale supercomputers. That being said, this study focuses on *intra-node* scalability to understand how multiple serial tasks (i.e., bootstraps) compete for local resources (cache, RAM, disk I/O) when a node is fully loaded. Both strong and weak scaling are examined.

Load balance, essential in any parallel and distributed system, is another key focus. Without assumptions, there is no guarantee that each bootstrap will take a similar amount of time for QCF fitting, making load balance a critical aspect to investigate.

Given the MapReduce-like execution profile of this code, we assume dynamic scheduling might be more suitable, provided the execution time outweighs scheduling overhead. Also, different scheduling strategies and load imbalance levels impact scalability, necessitating an examination of their effects in both dynamic and static scheduling.

We then construct a hybrid bootstrap distribution strategy, combining dynamic and static scheduling. Bootstraps are predetermined for each node. Each node then maintains a first-in-first-out (FIFO) task queue using Python's *multiprocessing* module. This hybrid mode allocates only one MPI rank per node, with all the local CPU cores as workers. Tasks are fetched from the queue as workers complete previous tasks, becoming idle only when no tasks remain.

While inter-node bootstraps are predetermined, the load imbalance is expected to be minor compared to the intra-node levels. Each node handles hundreds of bootstraps in production runs, ensuring statistically similar computational loads. Thus, this hybrid scheduling strategy appears to be sufficient for now.

As detailed in §IV-E, we conduct further analysis due to anomalies that appear in our standard testing of the code. In particular, we examine cache misses and CPU frequency variability when nodes are fully loaded. Throughout §IV, we present these unexpected results and, ultimately, our rationale regarding their causes.

IV. Results

Here we present the results of our computational experiments, enhancing our understanding of the target application's characteristics and improvements. §IV-A briefly explains our experimental setup. §IV-B addresses the overall time to solution, bottlenecks, and speedups, with bottlenecks re-implemented in C++. Comparisons between Python and C++ implementations are then made. §IV-C evaluates static versus dynamic scheduling for better parallelization. §IV-D examines strong and weak parallel efficiencies, critical for future deployment to exascale supercomputers. §IV-E investigates unexpected issues through cache usage and CPU frequency variations.

A. Experimental Set-Up

We conducted all of of our computational experiments on Tinkercliffs, a CPU-based cluster at Virginia Tech. Table I articulates the hardware specifications and software versions of the Tinkercliffs cluster. In addition, for any experiment shown, all the MPI ranks are evenly distributed between the two 64-core sockets in a node, totaling 128 CPU cores.

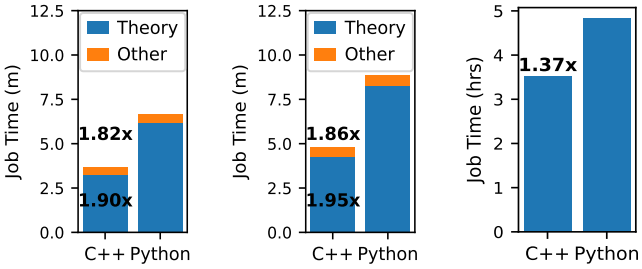
As noted earlier, we use `cProfile` to profile the execution of our code. In addition, we make use of Linux's `perf` tool and AMD's Performance Monitor Counter (*PMC* [13]) for detailed system information on cache usage and CPU frequency variability.

Hardware	Value	Software	Value
CPU	AMD 7702	GCC Version	13.2.0
Number of Cores	2x64	Python Version	3.10.12
System Memory	256 GB	Turbo Core	ON
Memory Bandwidth	350 GB/s	RAPL	ON

TABLE I: Hardware and software specifications

B. Characterizing Performance: Python and C++

We first profile the original Python code to identify bottlenecks and then re-implement the bottlenecks in C++ for fairness in comparison across languages. Fig. 3a clearly shows that the major bottleneck is in the theory module evaluation (see Fig. 1). In effectively porting the Python code to C++, we achieved a 1.82× speedup in the overall time to solution and a 1.90× speedup in the theory evaluation alone for a given bootstrap. In the Python implementation, the theory evaluation takes about 93.1% of the runtime, while in the C++ implementation, it takes 88.9%.



(a) Executing one bootstrap on one CPU.

(b) Executing 128 replicas of the same bootstrap on 128 CPUs. Each CPU executes one replica.

(c) Executing 128 different bootstrap on 128 CPUs. Each CPU executes one bootstrap.

Fig. 3: The overall time to solution and speedup after switching from Python to C++ for the theory module. Each bar represents the overall time-to-solution, distinguishing the theory execution portion from other operations. The speedups on top of the bars indicate overall speedup, while those in the blue portions reflect theory evaluation speedup. In the parallel setting (3b), each CPU executes the same bootstrap as in (3a), expected to show similar time-to-solution, which is not the case here. Figure 3c, depicts a more realistic situation where each CPU runs a different bootstrap. The bootstrap used in the first two sub-figures is also one of the 128 bootstraps here.

To determine if switching from Python to C++ for the theory evaluation affects scaling characteristics, we conduct a preliminary weak-scaling test. This test replicates the same bootstrap 128 times, each on a separate CPU core. Fig. 3b shows the job execution time for both implementations. The execution time per core matches the job time since all cores run the same bootstrap. As shown in Fig. 3b, the overall speedup and the theory module speedup are 1.86× and 1.95×, respectively. The theory evaluation takes about 94% of the runtime in Python and 89% in C++, aligning with the serial case and indicating consistent characteristics between Python and C++ when scaling.

However, when comparing Figs. 3a and 3b, we were surprised to find a significant difference in the overall time to solution for the Python implementation between the serial and parallel cases. Despite the simplicity of this weak-scaling test and using identical bootstraps, the Python implementation delivers a 34% *slowdown* in the parallel case. Furthermore, the C++ implementation also experiences a 31% slowdown. This unexpected behavior indicates the need for further scaling studies, as discussed in §IV-D.

Fig. 3c shows similar trending results as in Fig. 3b but with 128 *different* bootstraps, closer to a real-world use case. The specific bootstrap used in Fig. 3a is also included here.

The first notable observation in Fig. 3c is that the time scale is in *hours*, unlike the *minutes* observed in Fig. 3a, indicating significant load imbalance across the bootstraps. However, the extent of this imbalance is unclear from this test. Thus, we conduct further investigation with respect to load balance in §IV-C.

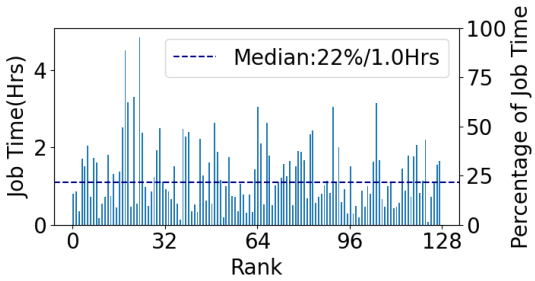
The second notable observation is that C++ with static scheduling provides only a 1.37× overall speedup, much lower than the 1.86× speedup in Fig. 3b. Intuitively, the speedup from the C++ implementation should remain consistent, regardless of load imbalance. That is, we expect a similar 1.86× speedup in Fig. 3c since all bootstraps should experience the same speedup. This discrepancy suggests an unexpected factor influencing performance and may relate to the slowdown observed from Fig. 3a to Fig. 3b.

C. Load Balance and Task Scheduling

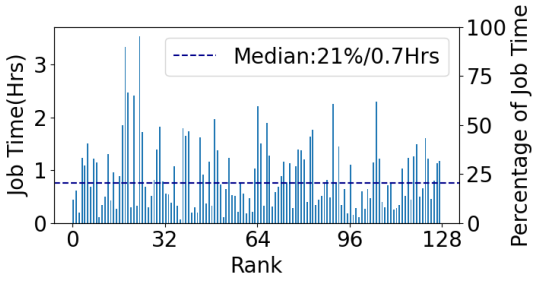
While the original code implementations, whether in Python or C++, suffer from load imbalance, the extent of the imbalance remains unclear. Fig. 4 shows the execution time of *each* bootstrap from Fig. 3c. Because each CPU core handles only one bootstrap, the run time equates to the busy time of each core. The longest-running bootstrap, rank 23, aligns with the overall time to solution in Fig. 3c. The overall CPU idle percentages of 78% for Python and 79% for C++ reveal significant load imbalance across CPU cores. This imbalance points to potential computational cycles wasted on idling.

Fig. 5 shows the probability of a bootstrap completing at different run times, normalized by the longest-running bootstrap's time. Since there is only one bootstrap per CPU core, the figure also indicates when most CPU cores become idle during the job's execution. Assuming all bootstraps achieve similar speedup after switching to the C++ implementation, the two curves were expected to be similar. However, Fig. 5 reveals a noticeable discrepancy, with the C++ implementation showing greater variance, indicating more divergence in bootstrap completion times. This implies a change in scaling behavior when switching from Python to C++, though the exact cause remains unclear. §IV-D on code scalability sheds some light on this change.

To address the load imbalance, we implement dynamic scheduling, as described in §III. Table II compares the time to solution and speedup between static and dynamic scheduling, using the original application code as the baseline. The results, derived from running 512 different bootstraps on 128 CPUs, show a 2.46× overall speedup from the original code.



(a) Execution time for the bootstrap on each CPU (Python theory evaluation).



(b) Execution time for the bootstrap on each CPU (C++ theory evaluation).

Fig. 4: Execution time of the bootstrap on each CPU core. Blue bars represent how much time a CPU core is busy. The left y-axis shows actual runtime in hours, while the right y-axis shows relative runtime to the longest-running bootstrap. A non-uniform bar height distribution indicates load imbalance. The median shows when 50% of CPU cores become idle. For example, in Fig. 4b, 64 CPU cores become idle after 0.7 hours or 21% of the longest-running task's execution time. Note that the x-axis label, MPI Rank, can be considered a CPU core's ID since ranks are bound to cores.

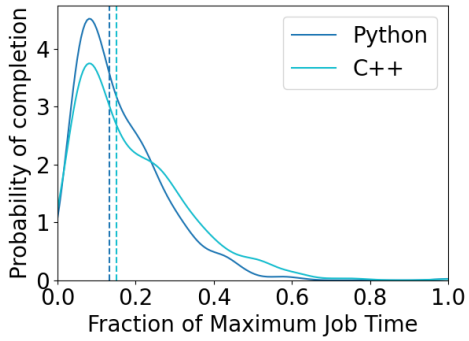


Fig. 5: Probability density distribution of execution time normalized to the overall job time. The curves represent the probability of a bootstrap completing at different times, normalized to the overall job time (i.e., the longest-running bootstrap).

Furthermore, the overall CPU idle-time percentage for the C++ implementation is now only 30%, down from 79% with static scheduling, resulting in a 62% improvement in CPU idle time. In the next subsection, we will examine the scalability characteristics of both scheduling approaches.

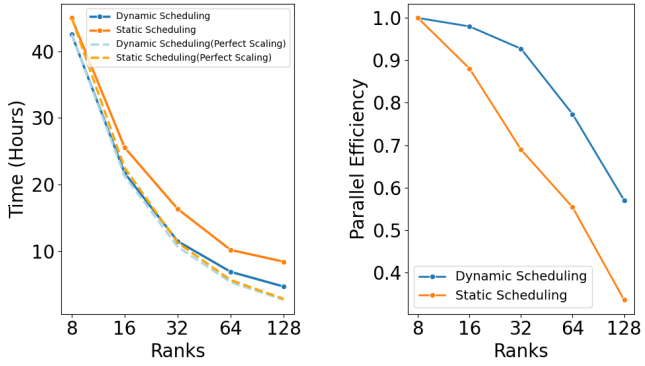
	Static (Python)	Static (C++)	Dynamic (C++)
Time (Hrs)	11.8	8.6	4.8
Speedup	1×	1.37×	2.46×

TABLE II: Time-to-solution and speedups for parallel execution of 512 different bootstraps using 128 CPUs and different scheduling approaches. The time to solution is defined as the job time. Note: Because each CPU core executes multiple bootstraps, the time to solution is not the execution time of the longest-running bootstrap.

D. Characterizing Scalability

For scalability, we perform both strong- and weak-scaling tests with both dynamic and static scheduling using our C++ theory implementation to understand the change in scaling characteristics when switching from static scheduling to dynamic scheduling.

We conduct a strong-scaling test by running 512 different bootstraps with varying numbers of CPU cores, up to 128 cores. However, due to computational costs, we were unable to perform 512 bootstraps with 1, 2, and 4 CPU cores, as these configurations exceed the time allocation of the Tinkercliffs cluster, a shared HPC cluster resource. Fig. 6 shows the time to solution and parallel efficiencies for both scheduling approaches, along with the curves for ideal scaling. The results indicate that dynamic scheduling offers better time to solution and better parallel efficiency.



(a) Time to solution. (b) Parallel efficiency.

Fig. 6: Strong scaling and parallel efficiency tests conducted with 512 bootstraps on 8 to 128 CPU cores. Parallel efficiency was calculated using the 8-core results as the baseline.

For static scheduling, the drop in efficiency appears to be due to the load imbalance. For dynamic scheduling, the steeper drop in efficiency begins around 32 CPU cores, also likely due to the load imbalance. When fixing the number of bootstraps but increasing the number of CPU cores in dynamic scheduling, each core handles fewer bootstraps, increasing the likelihood of deviations from the mean time to solution. Additional study and analysis are needed to determine whether further improving load balance can also improve strong scaling efficiency.

Figure 7 shows the results of the weak-scaling test, using two different setups: (1) adding random, different bootstraps

when doubling the number of CPU cores, mimicking a real-world use case and (2) adding identical bootstraps, maintaining a fixed computational load per unit.

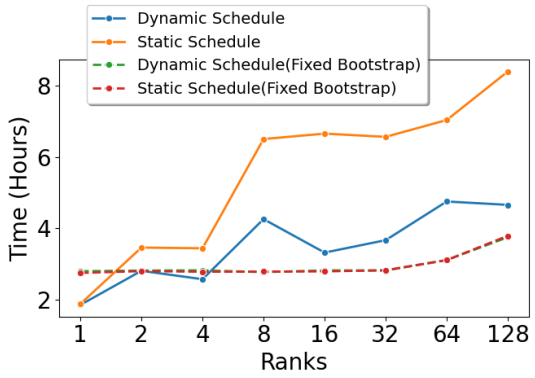


Fig. 7: Weak scaling in two different setups: (1) when doubling the number of CPU cores, the added bootstraps are random and different and (2) when doubling the number of CPU cores, the added bootstraps are exactly the same, i.e., "Fixed Bootstrap" in the legend.

The random-bootstrap test exhibits significantly poorer weak scaling for both scheduling approaches, as expected, due to the varying completion times for the bootstraps. Conversely, the fixed-bootstrap test demonstrates much better scaling with nearly perfect efficiency up to 32 cores for both approaches. This indicates that the scheduling overhead associated with dynamic scheduling is insignificant for this problem size. However, weak scaling performance drops significantly when scaling to 64 and 128 cores, suggesting an unknown factor affecting performance when using more than 50% of the CPU cores of a socket. This observation aligns with the slowdown noted in §IV-B, where executing the same bootstrap becomes slower when replicated and scaled to 128 CPU cores.

E. Memory Usage and CPU Clock Frequency

In this section, we seek to investigate the anomalies noted in the previous subsections:

- Slower overall execution of the same bootstrap when replicated and scaled to 128 CPU cores.
- Unexpectedly lower speedup for 128 different bootstraps with static scheduling when switching from Python to C++, compared to 128 replications of the same bootstrap.
- Differences in variance for bootstrap completion times between Python and C++ implementations.
- Poor weak scaling when using more than 50% of CPU cores.

We hypothesize that the aforementioned problems stem from (1) insufficient sharing of cache blocks, leading to increased cache miss rates and (2) reduced CPU clock rates.

For an AMD 7702 CPU in the Tinkercliffs cluster, each core has its own L1D and L2 cache, while the L3 cache is shared amongst four cores. Thus, to identify if poor caching behavior is responsible in part for inefficient scaling, we investigate the cache miss rates across the different cache levels (i.e., L1D, L2, and L3 cache misses) when running a fixed bootstrap with varying ranks.

Fig. 8 shows the average miss rate per rank and the normalized miss rate relative to a single rank. The L1D cache miss rate oscillates; the L2 miss rate drops until 32 ranks and then rises again; and the L3 miss rate remains consistent despite increasing ranks. Although interesting, this behavior does *not* explain the poor weak scaling beyond 32 CPU cores. Therefore, we assume the cache sizes of the AMD CPU are sufficient to prevent capacity misses for this problem.

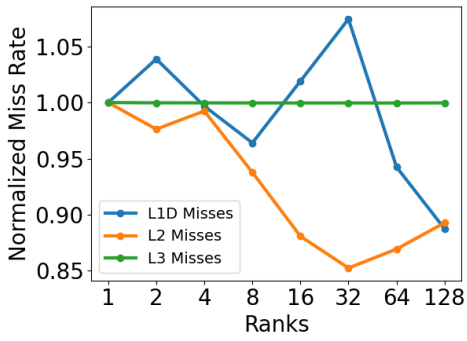


Fig. 8: Cache miss rates normalized against to that of using only one rank.

In Fig. 9, we compare the relationship between average CPU frequency and execution time as we vary the number of ranks. We observe that CPU frequency decreases proportionally with performance loss, explaining our poor scaling behavior. When accounting for the CPU frequency drop, the normalized execution time approximates linear weak scaling.

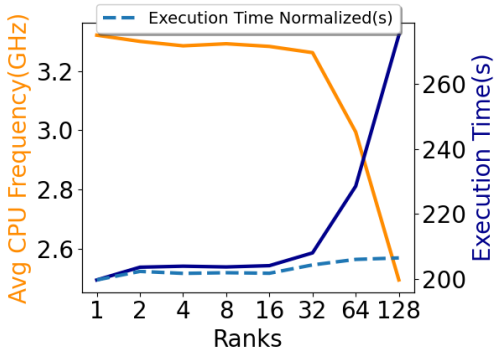


Fig. 9: Comparing execution time and average CPU frequency

The drop in frequency when CPUs are fully loaded answers the questions above. When there is load imbalance and all the CPUs are busy, the CPU frequency drops, causing all bootstrap executions to slow down. Once some bootstraps finish and CPU cores become idle, the CPU frequency bounces back (i.e., increases), speeding up the remaining bootstraps.

This phenomenon invalidates the assumption that all bootstraps receive the same C++ speedup. In the C++ implemen-

tation, more bootstraps finish earlier than in Python, meaning fewer bootstraps benefit from the CPU frequency bounce-back, resulting in lower speedups when comparing Fig. 3c to Fig. 3b. In addition, this explains the different variances in execution time between C++ and Python in Fig. 5.

The drop in CPU frequency is due to a combination of AMD's Turbo Core and Running Average Power Limit (RAPL) power management. With Turbo Core, the CPU is allowed to draw as much as 1.2× to 1.3× the nominal thermal design power (TDP) for *short* periods of time, meaning that the otherwise 2.0-GHz AMD 7702 CPU (e.g., " for all threads") can be clocked as high as 3.35 GHz (e.g., "for a single thread") for short periods of time. As a counterbalance, RAPL ensures that the long-term average power does not exceed the TDP and does so by reducing the CPU frequency and voltage to reduce power. Fig. 9 shows a peak CPU frequency at 3.35 GHz, thus verifying that Turbo Core was active. (The AMD 7702 CPU has a maximum boost frequency of 3.35 GHz).

V. Conclusion

This work focuses on the performance characterization and optimization of fitting quantum correlation functions (QCFs) to femtoscale experimental data, in preparation for running on exascale supercomputers. Re-implementing in C++ achieves a 1.8× speedup with balanced loads and 1.4× with significant load imbalance. Dynamic scheduling improves load balance, resulting a 62% percent of improvement in CPU idle time and a 2.46× overall speedup compared to the original application code. Dynamic scheduling also enhanced strong scaling efficiency, maintaining over 90% efficiency up to 32 cores. Both scheduling approaches performed well in weak-scaling tests up to 32 cores, with only a slight decline afterward.

Unexpected testing results reveal performance degradation with increased intra-node parallelism. This is due to the (typically) enabled power management in supercomputers (such as Tinkercliffs at VT) in the form of AMD Turbo Core and Running Average Power Limit (RAPL). With Turbo Core, the CPU may draw as much as 1.3× the nominal thermal design power (TDP) for *short* periods of time, meaning that the otherwise 2.0-GHz AMD 7702 CPU (e.g., " for all threads") can be clocked as high as 3.35 GHz (e.g., "for a single thread") for short periods of time. As a counterbalance, RAPL ensures that the long-term average power does not exceed the TDP and does so by reducing the CPU frequency and voltage to reduce power.

Specifically, to abide by AMD's RAPL long-term average power, the CPU frequency must necessarily drops when 50% or more of the CPU cores in order to not exceed the TDP, thus explaining the scaling efficiency drop and inconsistent C++ performance under load imbalance. Initially, all bootstraps suffer from the lower CPU frequency due to full node load. As CPU cores finish execution, the frequency bounces back, accelerating the remaining bootstraps. The difference in execution speeds between C++ and Python bootstraps affects the timing of this frequency bounce-back, resulting in different scaling characteristics between the two implementations.

Our study highlights the importance of analyzing single-node scaling behavior, as frequency drops at full load can be overlooked in node-based scaling studies. Future work includes developing features like detector simulation and automatic differentiability of mathematical calculations. These features will alter performance characteristics, necessitating further performance studies to ensure efficient deployment on exascale supercomputers.

References

[1] R. D. Ball *et al.*, "The path to proton structure at 1% accuracy," *Eur. Phys. J. C*, vol. 82, no. 5, p. 428, 2022.

[2] S. Bailey, T. Cridge, L. A. Harland-Lang, A. D. Martin, and R. S. Thorne, "Parton distributions from LHC, HERA, Tevatron and fixed target data: MSHT20 PDFs," *Eur. Phys. J. C*, vol. 81, no. 4, p. 341, 2021.

[3] T.-J. Hou *et al.*, "New CTEQ global analysis of quantum chromodynamics with high-precision data from the LHC," *Phys. Rev. D*, vol. 103, no. 1, p. 014013, 2021.

[4] R. D. Ball *et al.*, "The PDF4LHC21 combination of global PDF fits for the LHC Run III," *J. Phys. G*, vol. 49, no. 8, p. 080501, 2022.

[5] R. Boussarie *et al.*, "TMD Handbook," 4 2023.

[6] M. Diehl, "Generalized parton distributions," *Phys. Rept.*, vol. 388, pp. 41–277, 2003.

[7] Jefferson Lab, "JAM: Jefferson lab angular momentum Collaboration, https://www.jlab.org/theory/jam,."

[8] R. K. Ellis, W. J. Stirling, and B. R. Webber, *QCD and Collider Physics*, ser. Cambridge Monographs on Particle Physics, Nuclear Physics and Cosmology. Cambridge University Press, 1996.

[9] J. C. Collins, D. E. Soper, and G. Sterman, "Factorization of hard processes in qcd," in *Perturbative QCD*. World Scientific, 1989, pp. 1–91. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/9789814503266_0001

[10] A. Vogt, "Efficient evolution of unpolarized and polarized parton distributions with QCD-PEGASUS," *Comput. Phys. Commun.*, vol. 170, pp. 65–92, 2005.

[11] Y. Zhou, N. Sato, and W. Melnitchouk, "How well do we know the gluon polarization in the proton?" *Phys. Rev. D*, vol. 105, no. 7, p. 074022, 2022.

[12] Argonne National Laboratory, "QuantOm: QUAntum chromodynamics Nuclear TOMography Collaboration, https://www.anl.gov/phy/quantom,."

[13] AMD, "Preliminary Processor Programming Reference (PPR) for AMD Family 17h Model 31h, Revision B0 Processors," Advanced Micro Devices, Inc., Tech. Rep. 55803 Rev 0.91, Sep. 2020. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/55803-ppr-family-17h-model-31h-b0-processors.pdf