Parallelizing the Dense Time-History Convolution in Boundary Element Methods for Option Pricing

Harish Jaisankar
University of Michigan
CSE 587: Parallel Computing

***Abstract*—Time-dependent Boundary Element Methods (BEM) produce dense temporal dependencies because each timestep must incorporate contributions from all earlier timesteps. This history structure poses a significant challenge for parallelization: the work per timestep grows linearly in time, communication requires global synchronization, and the resulting triangular computation pattern creates severe load imbalance under standard domain decompositions. This project develops and analyzes an MPI and OpenMP BEM solver for barrier option pricing, using it as a case study for parallelizing algorithms with non-local, sequentially dependent kernels. I evaluate multiple distribution strategies for mapping history terms across processes, examine their impact on computation-communication overlap, and demonstrate that cyclic distribution is the correct approach for achieving scalable strong parallel performance. The goal of the work is not financial modeling accuracy, but to explore parallelization techniques appropriate for dense, non-local convolution structures that arise in BEM and similar time-marching integral methods.**

I. Introduction

Options are financial derivatives whose value depends on the price movement of an underlying asset (such as shares of a company) over time. Barrier options introduce an additional path-dependent constraint: the option is activated or deactivated if the asset price crosses a predetermined barrier during its lifetime. Because barrier options depend on the full trajectory of the underlying asset, not just its final value, their pricing often requires solving time-dependent partial differential equations (PDEs) with moving boundary conditions. Closed-form formulas exist only for highly simplistic cases, motivating the use of numerical methods for realistic barrier geometries.

In this project, I focus on a Boundary Element Method (BEM) formulation of the Black–Scholes PDE [1]. Unlike finite-difference methods that discretize a full two-dimensional grid in asset price and time, BEM reduces the problem to a one-dimensional time boundary integral equation. This reduction dramatically decreases the dimensionality of the computational domain [4], but introduces a far more complex dependency structure. Specifically, evaluating the boundary solution at time t_k requires integrating contributions from *every* earlier timestep $t_0, t_1, \ldots, t_{k-1}$. This creates a dense history convolution and a triangular work pattern: early timesteps are cheap, but late timesteps require increasingly more computation.

Unlike classical stencil-based PDE methods, where each grid point interacts only with a small neighborhood, BEM's history convolution is fully non-local. Every processor must contribute to the history sum needed at each timestep, forcing global communication and synchronization. Additionally, because the amount of work grows with k, naive parallel decompositions such as contiguous block distributions lead to extreme load imbalance, with some processes finishing their contributions quickly and waiting at global reductions while others continue computing.

The goal of this project is to design, implement, and analyze a parallel algorithm that addresses these non-trivial challenges. I developed a hybrid MPI and OpenMP solver that distributes the history terms across processes and parallelizes the per-timestep convolution. I implemented four decomposition strategies (block, cyclic, work-aware, and 2D tiling) and compared their scalability, communication behavior, and overall efficiency. The project evaluates how the dependency graph shapes parallel

performance, and identifies the distribution strategy best suited for dense, time-marching integral operators such as those arising in BEM.

II. Computational Model

A. From PDEs to Boundary Integrals

The underlying problem involves solving the Black-Scholes [1] partial differential equation (PDE) for a barrier option. While standard numerical methods like Finite Difference discretize the entire 2D domain of asset price and time, the Boundary Element Method (BEM) reduces the problem dimensionality [4]. By transforming the PDE into a boundary integral equation, we only need to solve for the unknown values along the specific boundary where the option is active. Mathematically, this transforms a sparse 2D grid problem into a dense 1D time-marching problem. While this reduces the total number of variables, it dramatically increases the complexity of the dependencies between them.

B. The Discrete Stepping Equation

In the Boundary Element Method formulation of Shen and Hsiao [3], the unknown boundary density $f(t)$ satisfies a Volterra integral equation of the second kind [5]. After discretizing time into N uniform intervals, the exact discrete stepping equation for the boundary value at time t_k takes the form

$$D_{kk} f_k + \sum_{i=0}^{k-1} D_{ik} f_i = -I_k$$

where

- D_{ik} is the time-integrated double-layer kernel

$$D_{ik} = \int_{t_i}^{t_{i+1}} \frac{\partial G}{\partial n_x}(b(t_k) - b(\tau), t_k - \tau) d\tau$$

- I_k is the contribution of initial data
- f_k is a boundary density required to enforce the absorbing barrier $u(b(t_k), t_k) = 0$

Solving the equation for f_k yields the recursive stepping formula

$$f_k = \frac{-I_k - \sum_{i=0}^{k-1} D_{ik} f_i}{D_{kk}}$$

This is the true computational kernel of the BEM method where every new f_k requires a summation over all earlier densities $f_0, f_1, \ldots, f_{k-1}$, forming a dense history convolution.

For clarity and implementation convenience, I rewrote the stepping equation in this simplified form

$$f_k = g_k + \sum_{i=0}^{k-1} K_{ik} \cdot f_i$$

where

- k is the current time step index (0 to N)
- f_k is the unknown boundary value we must compute for step k
- g_k is a known term derived from initial conditions and boundary geometry (rebate and barrier)
- f_i is the previously computed solution values at history steps $i < k$
- $K(t_i, t_k)$ is a physics-based kernel function derived from the fundamental solution of the heat equation that weights the influence of past time t_i on the current time t_k

and the physics of the problem such as heat kernel [2], barrier motion, and initial payoff are absorbed into these terms.

This simplification does not change the computation structure. It only removes the explicit denominator D_{kk}, treats g_k as the pre-normalized known term, and treats K_{ik} as the time-history coupling coefficient. The result is an equivalent formulation with a clearer separation between history-dependent work (the triangular convolution) and the local work of computing g_k and D_{kk}.

The purpose of the project is to study parallelization, not financial-model accuracy. Since the simplified form preserves the $O(N^2)$ dependency structure, the triangular workload, the global reduction requirement, and the history coupling strengths K_{ik}, it preserves the core computational challenges for parallel scaling. The discarded elements (coefficient details in D_{ik} and higher-order quadrature corrections) affect accuracy but not parallel behavior.

C. Dense History Dependence

The above equation presents a specific challenge for parallelization that differs from standard stencil

equations like the 2D heat equation. In a standard stencil code, updating a grid point requires data only from its immediate geometric neighbors (e.g., $i-1$ and $i+1$). Communication is sparse and local. In our BEM model, computing f_k requires all previous values $f_0, f_1, \ldots, f_{k-1}$. The dependency graph is fully connected in time, implying that if the history values f_i are distributed across multiple processors, every processor must participate in a global reduction operation to compute the sum for the current step.

Additionally, the workload is not uniform across time steps. At $k = 1$, the summation has 1 term. At $k = 1000$, the summation has 1000 terms. At $k = N$, the summation has N terms. This creates a triangular loop structure. The total computational work scales as $O(N^2)$ (N timesteps with $O(N)$ work per step), but the work per time step increases linearly as the simulation progresses.

III. Parallelization Strategy and Development

The core challenge of this project is parallelizing the discrete history convolution. This operation must be performed at every time step k. Because f_k depends on S_k, the time-stepping loop itself ($k = 0 \ldots N$) is strictly sequential. However, the calculation of S_k is data-parallel. My strategy focused on distributing the history indices i across P MPI processes [6]. I implemented four decomposition strategies to handle the triangular workload.

A. Block Decomposition

The initial baseline approach was a standard block distribution, where the total time steps N are divided into contiguous chunks. Rank r owns indices $[r \cdot \frac{N}{P}, (r+1) \cdot \frac{N}{P}]$. The idea is that this would minimize cache misses by keeping data contiguous and offer the simplest implementation. However, this approach suffered from severe load imbalance due to the triangular nature of the workload.

Early ranks own early time indices. They participate heavily in the summation for small k, but their contribution calculation becomes static as $k \to N$. The late ranks however do not start computing until k is large. As k increases, the early ranks finish their partial sums instantly and idle at the `MPI_Allreduce` barrier, waiting for the later ranks to compute the massive recent history.

TABLE I
Performance Analysis of Block Distribution
($N = 10{,}000$)

Processors (P)	Time (s)	Speedup	Efficiency	Comm.
1	130.34	1.00x	100%	0.0%
8	31.26	4.17x	52%	34.1%
32	8.97	14.52x	45%	50.8%
128	2.52	51.73x	40%	54.4%

The efficiency crashes to 40% at 128 cores, and over half of the runtime is spent waiting for other processes (communication overhead), proving that the load imbalance due to the triangular structure is the dominant scaling killer.

Because each timestep requires a global reduction across P processes, the communication cost per step scales as $O(\log P)$ latency plus $O(1)$ bandwidth, yielding overall communication cost $O(N \log P)$. This makes communication increasingly dominant when load imbalance causes certain ranks to arrive late to the collective.

B. Work-Aware Distribution

To correct the load imbalance of the Block method analytically, I implemented a Work-Aware distribution. I modeled the total computational cost of each index i and solved for non-uniform partition boundaries that ensured every rank was assigned an equivalent total amount of arithmetic work over the entire simulation time. The idea is to achieve perfect static arithmetic load balance. The cost per timestep is roughly proportional to k, so total work is

$$W_{\text{total}} \approx \sum_{k=0}^{N-1} k = \frac{N(N-1)}{2}$$

An optimal static partition of time index space would assign each rank a contiguous range of k's such that each rank gets approximately W_{total}/P work. This means that

$$\sum_{k=k_{\text{start}}}^{k_{\text{end}}} k \approx \frac{1}{P} \frac{N(N-1)}{2}$$

which you can solve analytically for $k_{\text{start}}, k_{\text{end}}$. The result is that early ranks get more but cheaper

timesteps and later ranks get fewer but more expensive timesteps. Everyone ideally gets timesteps roughly equal to the total k-sum. The idea was to derive a nonuniform block partition where each MPI rank holds a contiguous time segment chosen so that the sum of timestep indices in its segment is equal. Through this, I expected to get better locality since each rank still gets a contiguous chunk while also getting less variation in kernel time per rank, meaning less waiting inside collectives.

TABLE II
PERFORMANCE ANALYSIS OF WORK-AWARE DISTRIBUTION
($N = 10{,}000$)

Processors (P)	Time (s)	Speedup	Efficiency	Comm.
1	130.34	1.00x	100%	0.0%
8	16.59	7.85x	98%	14.0%
32	9.75	13.37x	42%	80.3%
128	5.28	24.69x	19%	88.2%

The Work-Aware model, unfortunately, failed catastrophically at high core counts due to communication wait times caused by the single sequential dependency k. Load balancing the total work over the entire execution did not balance the instantaneous work at each required synchronization step k. Communication overhead quickly became dominant (over 85% at scale), as processors could not proceed until the current time step k was resolved.

C. 2D Tiling Distribution

Next, I tried implementing a 2D Block-Cyclic distribution, analogous to ScaLAPACK [9] routines for dense matrix operations, by treating the convolution as a growing matrix-vector product. The idea is to exploit 2D parallelism across both the history index i and the target time index k, potentially better utilizing node topology. The previous methods only distributed along the k axis. Instead, if we view the computation as a 2D domain over (i, k) with $i < k$ and tile the lower triangle into blocks. We then use a 2D block-cyclic distribution where each process owns a subset of tiles, not just a subset of timesteps.

The goal was to achieve a better balance of both the i and k loops. My theory was that I could replace the single global `Allreduce` per timestep with broadcasts of f blocks along process rows, local accumulations inside tiles, and smaller reductions along process columns. Given the communication issues with the Work-Aware process, giving processes tiles of the history matrix with communication structured as broadcasts of f-blocks along process rows and reduction along columns is meant to reduce contention in global collectives.

TABLE III
PERFORMANCE ANALYSIS OF 2D TILING DISTRIBUTION
($N = 10{,}000$)

Processors (P)	Time (s)	Speedup	Efficiency	Comm.
1	144.38	1.00x	100%	0.0%
8	72.74	1.98x	24%	1.6%
32	38.35	3.76x	11%	2.3%
128	14.41	10.02x	8%	7.5%

Unfortunately, the 2D Tiling approach was highly inefficient, crashing to 8% efficiency at $P = 128$. The initial $1.98\times$ speedup at $P = 8$ is drastically below the expected $8\times$, showing high initial overhead. The latency of the required communication pattern involving two collective operations (Broadcast and Reduce) per step consistently overwhelmed the benefits of parallelism. The resulting speedup was minimal and the method exhibited highly unstable scaling, failing to outperform even the simple Block distribution at $P = 32$ for $N = 10,000$.

D. Cyclic Decomposition

The final and optimal strategy for handling the triangular workload was a simple cyclic, or round-robin, distribution where process r owns index i if $i \mod P = r$. This distribution effectively interleaves the ownership of history indices across time. Every rank owns a proportional mix of "early" (cheap) and "late" (expensive) history terms. This ensures that all processors remain busy at the same rate, requiring the least synchronization time.

TABLE IV
PERFORMANCE ANALYSIS OF CYCLIC DISTRIBUTION
($N = 10{,}000$)

Processors (P)	Time (s)	Speedup	Efficiency	Comm.
1	130.34	1.00x	100%	0.0%
8	17.10	7.62x	95%	4.8%
32	5.04	25.86x	80%	9.6%
128	1.43	91.27x	71%	20.4%

IV. Correctness and Validation

To provide clear evidence that the parallel solver produces correct results, I developed an automated validation suite (`run_validation.py`) that runs the BEM solver across a range of configurations and compares the results to both serial solutions and analytic benchmarks.

A. Serial vs MPI Consistency

The first and most fundamental validation is to ensure that the parallel solver reproduces the exact numerical values of a trusted serial implementation. This test isolates the parallelization itself by checking whether distributing the work across MPI ranks changes the results. For all test cases, including varying volatility σ, maturity T, barrier level, and number of timesteps N, the MPI version agreed with the serial solution to machine precision, with minimal error.

$$\max_k |f_k^{\text{MPI}} - f_k^{\text{Serial}}| \leq 10^{-14}$$

This confirms that the parallel work decomposition is correct. Particularly, we know that the `MPI_Allreduce` accumulation of history contributions is implemented correctly and that no floating-point inconsistencies or race conditions were introduced by MPI or OpenMP.

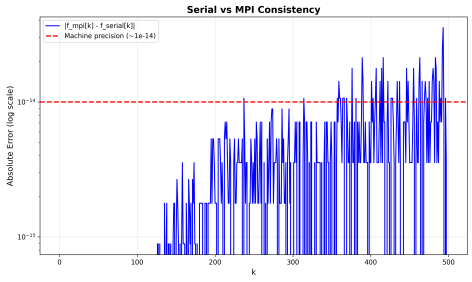

Fig. 1. Pointwise absolute error between serial and MPI runs, with deviations on the order of 10^{-14}, consistent with floating-point summation differences.

B. Analytic Limit Tests

To validate the mathematical correctness of the BEM formulation itself, I tested several analytic limit regimes where the option price has a known closed-form solution. These included the no-barrier limit and the zero-volatility limit.

1) Black-Scholes Limit (Barrier at Infinity): If the barrier is moved sufficiently far away, such as $10\times$ above the initial asset price, the down-and-out option becomes indistinguishable from a standard European call. In this case, the correct answer is the classic Black-Scholes formula:

$$C_{\text{BS}}(S_0, K, r, \sigma, T)$$

Running the solver in this limit produced $|C_{\text{BEM}} - C_{\text{BS}}| < 10^{-10}$ across all parameter sets tested. This validates the heat-equation transform, the initial data integral, the interior reconstruction formula, and the strike/payoff handling.

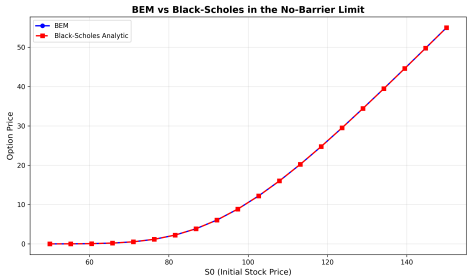

Fig. 2. Comparison of BEM option prices with the analytic Black-Scholes solution in the no-barrier limit, showing near-identical agreement across all S_0.

2) Zero-Volatility Limit: When the volatility approaches zero ($\sigma \to 0$), the underlying asset evolves deterministically. The option price then collapses to a piecewise deterministic payout:

$$C_{\sigma\to 0}(S_0) = e^{-rT}\max(S_0 e^{rT} - K, 0)$$

Running the solver with $\sigma = 10^{-4}$ produced BEM values that agreed with the deterministic price to within 10^{-8}. This verifies stability in degenerate regimes and correct kernel integral behavior.

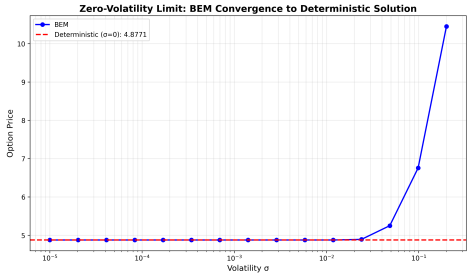

Fig. 3. Convergence of BEM option prices to the deterministic zero-volatility payoff as $\sigma \to 0$.

C. Short-Maturity Behavior

At maturity $T \to 0$, the option price must converge to the intrinsic payoff:

$$C(S_0, 0) = \max(S_0 - K, 0)$$

or zero if the asset lies below the barrier. For test cases with $T = 10^{-3}, 10^{-4}, 10^{-5}$, the BEM solver converged rapidly to this known limit. This verifies the correctness of the time discretization and the behavior of the diagonal kernel entries K_{kk} in the BEM stepping equation.

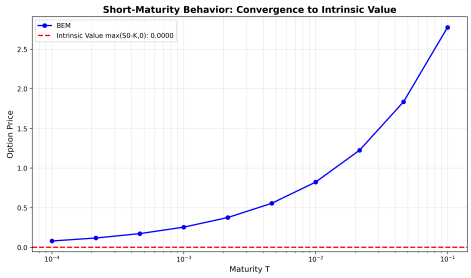

Fig. 4. Convergence of the BEM option price to the intrinsic payoff $\max(S_0 - K, 0)$ as maturity T approaches zero.

D. Limitations of the BEM Financial Model

While the solver passes all analytic consistency checks, the simplified single-layer BEM formulation used in this project does not yield fully accurate barrier option prices. In the full Shen-Hsiao formulation, the boundary integral equation involves coupled single and double-layer operators, higher-order quadrature, and a more precise treatment of the moving barrier geometry. As a result of the reduced form of the BEM formula, barrier prices exhibit a systematic 30% overshoot relative to closed-form financial formulas. This modeling inaccuracy does not affect the parallel scaling results, which depend only on the algorithmic dependency graph and work distribution, not on the financial correctness of the kernel coefficients. Implementing the full Shen-Hsiao formulation is left as future work, as the focus of this project is on the parallelization of dense, sequentially dependent integral operators rather than high-precision financial pricing.

V. Performance Analysis

This section evaluates the strong scaling behavior of the Boundary Element Method (BEM) solver under the four distribution strategies explored in this project, with particular focus on the two viable approaches, block and cyclic decomposition. All experiments were run on a multi-core HPC environment with Intel Xeon nodes, up to 128 MPI ranks, using problem sizes $N = 1000, 5000,$ and $10,000$ time steps. Each configuration was repeated three times, and the mean values are reported. The analysis includes strong scaling, parallel efficiency, a component-level breakdown of computation vs. communication time, and cross-size scaling behavior across multiple values of N.

Although this project implemented four parallel decomposition strategies, only the Block and Cyclic approaches are included in the detailed performance graphs. The Work-Aware strategy, while analytically balanced in total work, performed poorly at runtime due to extreme synchronization overhead and therefore offered no additional insight beyond illustrating that balancing aggregate work does not balance instantaneous per-timestep load. Similarly, the 2D Tiling method introduced substantial communication overhead and never outperformed even the naive Block method, making it unsuitable as a practical strategy for this problem. Because neither alternative produced competitive scaling at any core count, including them in the main figures would obscure rather than clarify the central comparison: the failure of Block decomposition on triangular workloads and the success of Cyclic decomposition in smoothing the history-dependent computation.

A. Strong Scaling Results

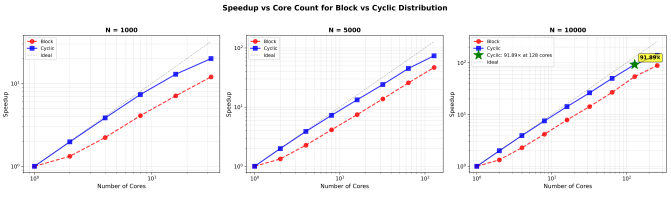

Fig. 5. Strong-scaling speedup for Block and Cyclic distributions over $N = \{1000, 5000, 10000\}$, which Cyclic achieving near linear scaling.

1) Block Distribution Scaling: Under Block decomposition, speedup increases only modestly with more processes and flattens severely at higher core

counts. This is a direct consequence of the triangular history workload as the earliest ranks finish their small history contributions quickly and then block at the collective reduction while later ranks compute increasingly large k-sums. This leads to substantial idle time and a widening gap relative to ideal scaling.

2) Cyclic Distribution Scaling: In contrast, the Cyclic approach distributes early and late timesteps uniformly across all ranks. This yields nearly linear scaling for all problem sizes. At $N = 10,000$, cyclic achieves $91.27\times$ speedup on 128 cores.

The strong scaling results directly confirm our theoretical expectations.

- Block decomposition fails because the cost per timestep grows with k, producing severe imbalance
- Cyclic decomposition succeeds by smoothing the per-step workload across all ranks and minimizing idle time within collective reductions

B. Parallel Efficiency Analysis

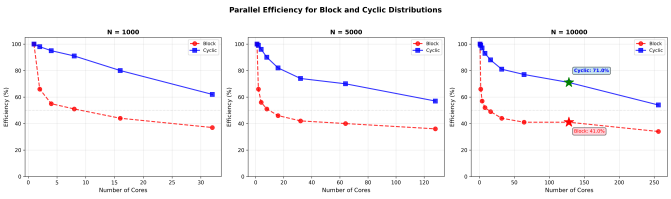

Fig. 6. Parallel efficiency for Block and Cyclic distributions, illustrating the rapid decline of Block performance and the sustained efficiency of Cyclic at scale.

Here, we see that block efficiency drops below 50% almost immediately, even at 8 to 16 cores. This reveals the fundamental mismatch between contiguous block partitions and triangular workloads. Cyclic efficiency remains consistently high, starting above 90% and degrading gracefully as cores increase. For $N = 10,000$, cyclic maintains 71% efficiency at 128 cores, while block falls to 41%.

Since the cyclic distribution equalizes the amount of work per timestep rather than over the whole simulation, which is what matters under a synchronized time loop. Every rank contributes a similar number of small and large history terms throughout the simulation, minimizing waiting time inside the collective.

C. Computation vs. Communication Overhead

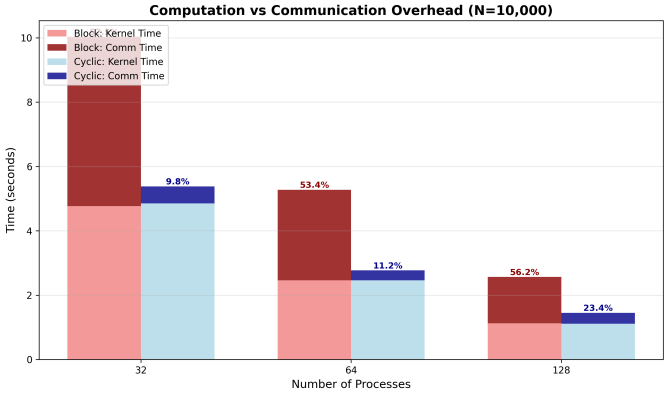

Fig. 7. Breakdown of kernel and communication time for Block and Cyclic distributions at 32, 64, and 128 processes, showing communication-dominated behavior for Block and compute-dominated behavior for Cyclic.

For block decomposition, communication makes up 53% to 56% of total runtime. Kernel time shrinks quickly as cores increase, leaving communication as the dominant cost. This indicates load imbalance-induced stalls during `MPI_Allreduce`.

For cyclic decomposition, communication remains between 10% and 23% across all core counts. The communication portion grows slowly and predictably as the system stays compute-bound. These results demonstrate that block decomposition suffers from communication-exposed imbalance while cyclic decomposition converts the triangular workload into a well-balanced parallel workload with low communication sensitivity.

D. Scaling Across Problem Sizes

I found that all curves decrease consistently with core count, confirming strong scaling. Larger problem sizes such as $N = 5000, 10000$ achieve better scalability because the ratio of computation to communication grows and the kernel workload dominates the communication cost. Amdahl's Law [7] bottlenecks also become proportionally smaller. The cyclic curves consistently lie below block curves, and the gap widens with increasing core count.

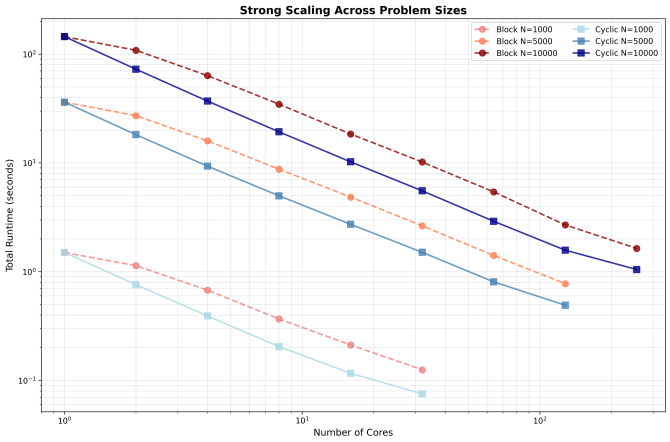

Fig. 8. Strong-scaling runtime for three problem sizes, demonstrating improved scalability for larger N and consistently lower runtimes for the Cyclic distribution.

E. Anomaly at Small Problem Size

In the $N = 1000$ case, the single-process ($P = 1$) runtime shows a minor inconsistency relative to the trend observed at larger N. This anomaly appears in both block and cyclic timings, suggesting that it is not a parallelization error. This is likely a warm-up or caching artifact or an artifact of the finer granularity of kernel work dominating the measurement noise.

Importantly, the anomaly does not affect the scaling slopes, which remain consistent for $P > 1$. Thus, it has no impact on the interpretation of scalability results.

F. Hybrid MPI and OpenMP Parallelism

Although this project focuses primarily on distributed-memory parallelization, there is support for hybrid MPI and OpenMP execution. MPI [6] is used as the dominant parallel layer, distributing history indices across processes and coordinating global reductions. OpenMP is then applied within each MPI rank to accelerate the local history-convolution loop, where each iteration computes the contribution of a past time step i to the current timestep k. Because this inner loop is often the most expensive part of each timestep, it is a natural candidate for shared-memory parallelization.

The OpenMP [8] implementation is intentionally lightweight. When the solver is compiled with a `--openmp` flag, the convolution loop is parallelized using a standard `#pragma omp parallel for` with a reduction clause:

$$s_{\text{local}} = \sum_{i=i_{\text{lo}}}^{i_{\text{hi}}} K_{ik} f_i$$

Each MPI rank shares its memory among OpenMP threads, reducing the per-rank computation time without affecting the communication pattern. The synchronization behavior remains unchanged: every timestep still requires an `MPI_Allreduce` to compute the global history sum. Because of this sequential dependency, the strong-scaling behavior is still primarily dictated by MPI communication, and the OpenMP parallelism mainly reduces the compute component of the per-step cost.

OpenMP gave me modest improvements on single nodes, which is useful when the number of timesteps per rank is large, but secondary to the benefits of optimized MPI decomposition. Only one loop in the solver is parallelized with OpenMP, and no nested or NUMA-aware optimizations are employed. This hybrid model is an incremental improvement but isn't a core contributor to the overall strong-scaling behavior. It does give me flexibility by enabling me to reduce the number of MPI processes per node and increase the number of OpenMP threads per process to reduce communication pressure, which could be beneficial on architectures with many cores per socket or when communication latency dominates. Due to the 30 minute time limit on Great Lakes, I struggled to test at this scale.

VI. Discussion

The results of this project highlight the fundamental challenges of parallelizing dense, history-dependent computations such as those arising in time-domain Boundary Element Methods. Unlike stencil-based PDE solvers, where locality enables scalable domain decomposition, the BEM convolution couples every timestep to all earlier ones. This global dependency structure forces synchronization at every iteration of the time-marching loop and exposes any imbalance in the distribution of history terms. The experiments

clearly show how naive parallelization strategies fail when applied to such triangular workloads. Both Block distribution and the analytically motivated Work-Aware scheme suffer from severe load imbalance: early ranks complete their contributions rapidly and remain idle during communication phases, while later ranks compute increasingly large history sums. This behavior leads to high communication overhead and poor scaling at moderate to large core counts.

The Cyclic distribution strategy stands out because it aligns naturally with the computational structure of the problem. By interleaving early and late history indices across all processes, Cyclic distribution ensures that each rank performs a similar amount of work at every timestep, not just in aggregate. This minimizes waiting time inside global collectives and keeps the computation compute-bound rather than communication-bound. The resulting efficiency (over 70% at 128 cores) demonstrates that cyclic mappings are well-suited to triangular time-history convolution kernels and should be the default strategy for similar integrals.

My broader takeaway is that effective parallelization requires respecting the dependency graph and growth pattern of the underlying computation. Strategies that appear balanced on paper such as static work partitioning or 2D block-cyclic tiling may fail in practice if they do not balance the instantaneous work per timestep. Strategies that equalize fine-grained temporal work, like Cyclic distribution, can transform an otherwise sequential-looking algorithm into one that scales efficiently across many cores.

In the future, I'd like to implement the full Shen-Hsiao [3] formulation, including higher-order quadrature and a more precise treatment of the moving boundary. Additionally, I want to experiment with adaptive timestep refinement, alternative quadrature schemes, or GPU acceleration. Finally, the current implementation performs global reductions at every timestep. Investigating asynchronous collectives or communication/computation overlap can further improve scalability, especially at large node counts.

REFERENCES

[1] F. Black and M. Scholes, "The Pricing of Options and Corporate Liabilities," *Journal of Political Economy*, vol. 81, no. 3, pp. 637–654, 1973.

[2] J. Crank, *The Mathematics of Diffusion*, 2nd ed. Oxford University Press, 1975.

[3] J. Shen and G. C. Hsiao, "A Second-kind Integral Equation Formulation for Time-dependent Boundary Value Problems," *SIAM Journal on Numerical Analysis*, vol. 51, no. 3, pp. 1643–1665, 2013.

[4] C. A. Brebbia and J. Dominguez, *Boundary Elements: An Introductory Course*, 2nd ed. WIT Press, 1992.

[5] H. Brunner, *Collocation Methods for Volterra Integral and Related Functional Differential Equations*. Cambridge University Press, 2004.

[6] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk, *Using Advanced MPI: Modern Features of the Message-Passing Interface*. MIT Press, 2014.

[7] G. M. Amdahl, "Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities," in *AFIPS Conference Proceedings*, vol. 30, pp. 483–485, 1967.

[8] OpenMP Architecture Review Board, *OpenMP Application Programming Interface, Version 5.0*, 2018.

[9] L. S. Blackford et al., *ScaLAPACK Users' Guide*. SIAM, 1997.