Parallel Black-Scholes Solver on Adaptive Sparse Grids

Sneha Jaiswal, Anushka Parmanand
https://418-cmu-26.github.io/

1 Summary

We present a parallel solver for the multi-asset Black-Scholes PDE using OpenMP task-based parallelism on adaptive sparse grids. Pricing basket options on $d \geq 2$ correlated assets requires solving a d-dimensional PDE numerically; sparse grids reduce the node count from $O(N^d)$ to $O(N \log^{d-1} N)$ while preserving accuracy, but their hierarchical structure demands specialized parallel algorithms. The dominant computational cost is a matrix-vector product applied at every conjugate gradient iteration, which we parallelize via a recursive up-down scheme that exposes two independent layers of work: $d + 1$ concurrent passes per iteration and branching task trees within each pass. We further optimize memory access through a Compact storage layout that sorts nodes by global level, eliminating pointer indirection and improving cache utilization over a HashMap-implementation baseline. We evaluate four configurations combining two storage schemes (HashMap, Compact) with two grid topologies (Regular, Adaptive) across dimensions $d = 3$ to 6 on GHC and PSC machines. On GHC 8-core machines, Compact + Regular achieves up to 6.7× speedup; on PSC machines with 64 threads, Compact + Adaptive reaches roughly 30× at $d = 6$. Cache miss profiling confirms that Compact storage reduces cache misses relative to HashMap across all configurations, and TAU profiling identifies OpenMP scheduling overhead and serial CG bookkeeping as the primary remaining bottlenecks.

2 Background

A financial option gives the holder the right to buy or sell a stock at a fixed price K (the *strike price*) at some future date. The Black-Scholes equation is a PDE whose solution $u(S, t)$ gives an option's price as a function of the current stock price S and time t. For a single stock, the Black-Scholes PDE has a known closed-form solution. But this is not the case when the option depends on $d \geq 2$ stocks simultaneously (a *basket option* — think an option on a portfolio of assets). In such cases, the PDE has d spatial dimensions, one per stock price, and must be solved numerically on a grid.

Thus, the core data structure supporting a discretized Black-Scholes solver is a multi-dimensional grid. However, regular Cartesian grids with N points per dimension suffer the curse of dimensionality, since they need N^d total points, which is computationally infeasible. In contrast, **sparse grids** include grid points more selectively at each dimension. Using hierarchical basis functions, a sparse grid starts from a very coarse grid covering the whole domain and iteratively adds finer grid points where needed. The degree of granularity is denoted using *levels*, and each sparse grid point is represented by d-length level and index vectors $(\mathbf{l}, \mathbf{i})$. At level l, indexes i are odd values in the range $0 < i < 2^l$, so higher levels correspond to finer granularity. Additionally, subspaces with bad cost-benefit ratio are omitted: specifically, the input parameter n restricts the sparse grid to points that satisfy $|\mathbf{l}|_1 + d - 1 \leq n$, where $|\mathbf{l}|_1$ denotes the global level (level vector sum). Since the Black-Scholes PDE is smooth over most of the domain, finer resolution is required only where the solution is changing rapidly. Thus, sparse grids reduce the number of grid points from $O(N^d)$ to roughly $O(N \log^{d-1} N)$ while losing minimal accuracy.

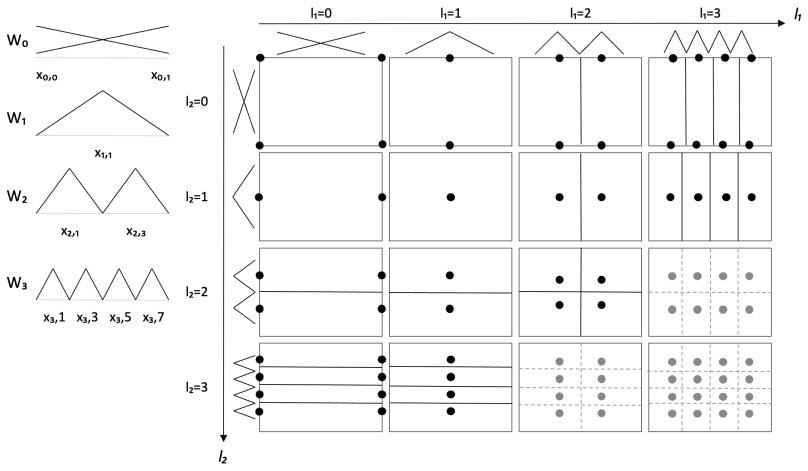

Figure 1: Diagram of how sparse grid is structured

The algorithm operating on this data structure in order to price a basket option can be broken down into a pipeline with four main steps:

1. Principal Axis Transformation (PAT). Directly discretizing the Black–Scholes PDE is numerically difficult due to the variable coefficients and correlated dimensions. This transform reverses the time ($\tau = T - t$), substitutes log-prices ($x_i = \log S_i$), and performs an eigendecomposition of the covariance matrix. Rotating coordinates by the eigenvector matrix decouples all dimensions, and a further drift-eliminating translation and discounting substitution yields a d-dimensional heat equation:

$$\frac{\partial u}{\partial \tau} = \frac{1}{2} \sum_k \lambda_k \frac{\partial^2 u}{\partial z_k^2}$$

   where λ_k are the eigenvalues of Σ. The computational cost of this step is dominated by the $O(d^3)$ eigendecomposition, performed once at initialization.

2. FEM discretization and linear system. We then discretize using the finite element method (FEM) with the hierarchical hat functions. This yields a semi-discrete system. $B\dot{u} + Lu = 0$ where B is the mass matrix and $L = \frac{1}{2}\sum_k \lambda_k C_k$ is the stiffness/Laplacian matrix. Then discretizing in time with Crank–Nicolson produces a linear system at each step. $(B + dt \cdot L)\, u^{n+1} = (B - dt \cdot L)\, u^n$. This is solved iteratively with the conjugate gradient (CG) method. Boundaries are handled by separating out a precomputed boundary function b_0 and adjusting the right-hand side at each step. This never assembles the B or L matrices explicitly and instead applies them via matrix–vector products.

3. Matrix–vector product. The linear system from step 2 is solved iteratively with the conjugate gradient method. Each CG iteration requires one matrix–vector product $(B + dt \cdot L) \cdot p$ and runs for up to hundreds of iterations per time step. This is a dominant cost of the solver. Both B and L decompose into tensor products of one-dimensional operators via the up–down scheme, so the product can be applied without ever constructing the sparse matrix explicitly. We aim to apply parallelism here. B requires one up–down pass with mass operators in every dimension, and each of the d terms in L require an independent

pass with the stiffness operator in dimension k and mass operators in all others. This yields $d + 1$ independent up–down passes per CG iteration.

4. Hierarchization. Before the time-loop, the payoff function must be converted from nodal values at grid points to hierarchical surplus coefficients. This proceeds level–by–level from coarse to fine as the surplus at each node is dependent on the node's lower-level parents.

3 Parallel Characterization of the Algorithm

3.1 Data dependencies

The hierarchization step and up-down pass exhibit a strict level-by-level dependency structure. In hierarchization, the surpluses at high level nodes depend on lower level surpluses, so levels must be processed coarse-to-fine in sequence. Within a single level, however, all node surpluses are mutually independent as no two nodes with the same global level sum share a parent-child relationship in any dimension. The up-down algorithm has a similar structure. The upward (fine-to-coarse) mass sweep propagates contributions from children to parents, so a coarser level's accumulated value is not final until all finer levels have been processed. Conversely, the downward (coarse-to-fine) pass distributes values from parents to children, and each level waits on the one above it. Within any given level, nodes are mutually independent.

3.2 Available Parallelism

The $d + 1$ up-down passes required per CG iteration (one for the B matrix and one for each of the d eigenvalue-weighted stiffness terms) read the same input vector $\mathbf{p}$ and write entirely disjoint output vectors. Therefore, they constitute the outermost, fully independent layer of parallelism and are handled as concurrent OpenMP tasks in `apply_matvec_parallel`. Within each pass, the recursive up-down structure exposes a second layer of parallelism: the b branch and c branch. The b branch is an up sweep followed by recursive descent and the c branch is recursive descent followed by a down sweep. These write to disjoint vectors, and only require synchronizing at the join point. Finally, within a single sweep at a fixed level, all nodes are independent and can be processed in parallel.

3.3 Data Locality

The algorithm is partially data-parallel within a level (as the same arithmetic is applied independently to each node but across levels there are sequential dependencies). Thus, locality of memory accesses is important. All nodes at a given level are accessed together during a level sweep, so they should ideally be contiguous in memory.

4 Approach

We implemented the solver in C++ using OpenMP for shared-memory parallelism. While our approach closely follows Heinecke et al.'s implementation, all code is original. The key algorithmic target for parallelism is the matrix-vector product performed at every step of the CG solver, specifically the `apply_matvec_parallel` function which computes $Bx + dt \cdot Lx$ for a given search direction.

We targeted the GHC and PSC shared-memory machines using OpenMP for a few reasons. First, the recursive task model of `updown_recursive` maps naturally onto OpenMP's task-based

parallelism: each recursive call spawns two independent branches, and OpenMP's scheduler assigns them to idle threads dynamically without programmer intervention. Second, the level-by-level dependency structure of the sparse grid, where coarser levels must complete before finer ones, makes GPU impractical as a CUDA kernel cannot easily enforce this ordering across warps without global synchronization. This would serialize execution and eliminate the benefit of the GPU. Third, the irregular node indexing in the adaptive topology would cause warp divergence on a GPU, since nodes at the same level-sum may have very different child counts after refinement. The GHC and PSC machines' shared address space also allows all threads to read the same input vector α without any data transfer overhead, which is critical given that `apply_matvec_parallel` performs $d+1$ reads of the same vector per CG iteration. In contrast, an MPI-based parallel implementation would involve heavy communication costs, since each parallel unit would store the input and output vectors in local private memory; thus the interconnect would be overwhelmed by both inter-thread synchronization traffic and per-thread memory traffic.

Our implementation evolved through four iterations. Initially, we implemented the up-down scheme iteratively using `#pragma omp parallel`, but this structure left half of the available parallelism unexploited. In our milestone, we report speedup ratios of 1.4×, 1.8×, and 0.02× with 2, 4, and 8 threads respectively for this iterative version. This motivated our second iteration: a recursive task-based implementation.

Our third iteration optimized the sparse grid storage implementation, addressing storage overhead and low cache utilization. Programming this 'Compact' storage scheme required complete restructuring of our sparse grid, including a new interface, and also took several attempts attempts to implement correctly. Finally, our fourth iteration introduced adaptive refinement of the sparse grid to improve solution accuracy and stress test using larger, irregular inputs. These iterations are detailed below.

4.1 Up-Down Scheme

1. B and L decompose into tensor products of 1D operators, avoiding explicit matrix construction entirely.
2. A single up-down pass in dimension k consists of two sweeps:
   (a) The pass alternates between a mass operator and a stiffness operator depending on whether dimension k is the gradient dimension. In non-gradient dimensions, the mass operator is applied; in the gradient dimension, the stiffness operator is applied instead.
   (b) Upward sweep (fine-to-coarse): for the mass operator at level l_k, each node sends $\frac{2}{3}h_k\alpha_j$ to itself and $\frac{1}{3}h_k\alpha_j$ to its parent.
   (c) Downward sweep (coarse-to-fine): for the mass operator, each node sends $\frac{1}{3}h_k$ to each child. For the stiffness operator, each node contributes $\frac{1}{2}2^{l_k}\alpha_j$ to itself and $-\frac{1}{4}\alpha_j$ to each child.
3. To apply $B + \Delta t \cdot L = B + \Delta t \cdot \frac{1}{2}\sum_k \lambda_k C_k$, we compute $B\alpha$ and $L\alpha$ separately. B requires one up-down pass with the mass operator in every dimension, and L requires d independent passes, one per eigenvalue-weighted stiffness term. The $d+1$ total passes all read the same input vector α and write to disjoint output vectors, making them fully independent.

4. `updown_serial` processes levels sequentially within each sweep. `updown_recursive` instead uses OpenMP tasks to split into two independent branches at each recursive level:
   (a) The b-branch performs an upward sweep on α and recurses on the result α_{up}.
   (b) The c-branch recurses on the original α and then applies a downward sweep to the result.
   (c) Both branches write to disjoint vectors b and c, which are marked `shared` to avoid per-task copies, and are merged after a `taskwait` barrier.
5. `apply_matvec_parallel` spawns all $d{+}1$ top-level calls to `updown_recursive` as concurrent tasks, constituting the outermost layer of parallelism.
6. Two serial limits remain: the $d = 1$ base case is fully serial since no further branching is possible, and within any single sweep, levels must be processed in dependency order — upward sweeps from fine to coarse, downward sweeps from coarse to fine.

In the very first version of this solver (described in Milestone report), we implemented the Up-Down scheme iteratively rather than recursively. While this approach allowed us to take advantage of the `#pragma omp parallel for` directive and thus test both static and dynamic scheduling, it did not support the full extent of parallelism allowed by the sparse grid's hierarchical structure. In particular, the iterative structure did not allow independent up and down sweeps (the b and c branches) to run in parallel.

Thus, our second iteration parallelizes the recursive Up-Down algorithm:

```
1:  function UPDOWN_RECURSIVE(α, d_cur, grad_dim)
2:      allocate b[N], c[N] ← 0
3:      if d_cur > 1 then
4:          #pragma omp task shared(b)                    ▷ b branch – independent
5:              α_up ← up_sweep(α, d_cur, grad_dim)
6:              b ← UPDOWN_RECURSIVE(α_up, d_cur − 1, grad_dim)
7:          #pragma omp task shared(c)                    ▷ c branch – independent
8:              mid ← UPDOWN_RECURSIVE(α, d_cur − 1, grad_dim)
9:              c ← down_sweep(mid, d_cur, grad_dim)
10:         #pragma omp taskwait          ▷ barrier: merge requires both branches
11:     else                                          ▷ base case d_cur = 1: serial
12:         b ← up_sweep(α, 1, grad_dim)
13:         c ← down_sweep(α, 1, grad_dim)
14:     end if
15:     for all i do b[i] += c[i]
16:     end for
17:     return b
18: end function
19:
20: function APPLY_MATVEC_PARALLEL(α, λ)
21:     #pragma omp task shared(result)
22:         result_k ← UPDOWN_RECURSIVE(α, d, k)
23:         result_k ×= λ_k     (for k ≥ 1)
24:     return result_0 + Δt · Σ_{k=1}^{d} result_k
25: end function
```

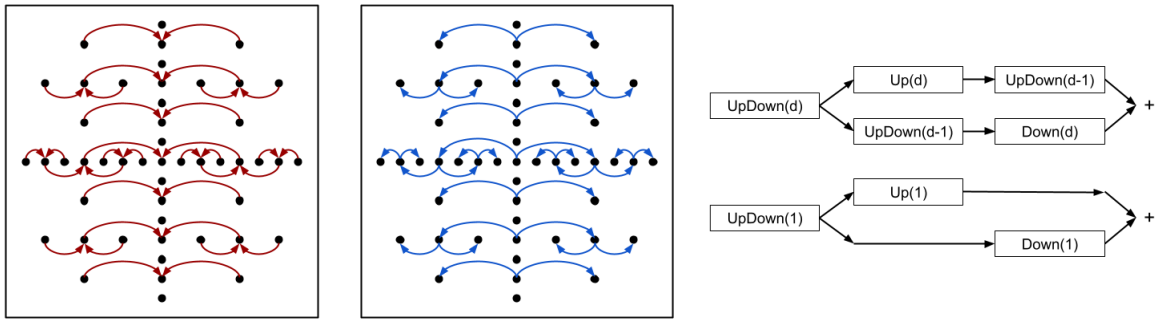

Figure 2: Traversal of up-operation (left) and the down-operation (middle) in the first dimension of a two-dimensional sparse grid. Algorithmic structure of `updown_recursive` (right), splitting into b & c branches and recursing in dimension d.

From a code-to-hardware perspective, the flat node vector is allocated once in shared memory and accessed by all threads without copying in `updown_recursive`. The $d + 1$ independent up-down passes in `apply_matvec_parallel` become $d + 1$ top-level OpenMP tasks, one for the mass matrix B and one per eigenvalue-weighted stiffness term, saturating the thread pool immediately at the outermost level. Within each pass, the `b` and `c` branches are concurrent and individually parallelized. Since the function is called recursively, breaking down the workload into tasks is not obvious (unlike a for loop, for example, where each iteration translates into a task). The `#pragma omp task` allows each recursive level to spawn two independent tasks (one per branch), building a task tree, and add them to OpenMP's work queue. Thus, the work scheduling is inherently dynamic since tasks in the queue are assigned to threads as they become available and each thread operates on its assigned node ranges independently. At each recursive level, the number of tasks doubles, producing roughly 2^d total per call to `updown_recursive`: this is ideal for dynamic scheduling, since high "parallel slack" (more independent work than parallel execution capability) allows better workload balance over threads.

Additionally, `shared` tells OpenMP to avoid creating private copies of common vectors at each thread, which prevents the memory footprint from scaling with the (large) number of tasks. The `#pragma omp taskwait` directive enforces a synchronization barrier: all tasks across both the `b` and `c` branches must complete before their results can be merged together.

4.2 Grid Storage

In the third iteration of our parallel Black-Scholes solver, we optimized the sparse grid implementation. All nodes (grid points) are stored in a flat 1D vector called `nodes`, but the algorithm requires accessing nodes by their global level sum or by their unique $d-$length level and index vectors (for example, when fetching a given node's parent or children across levels).

For ease of use, our initial sparse grid implementation relied on large auxiliary data structures: an unordered `node_map` and a 2D vector `level_buckets`. Using special `NodeKey` structs to represent a combination of level vector and index vector, `node_map` stores each node's 'id' – its index within the flat storage. Then, `level_buckets[s]` contains a vector of all ids corresponding to nodes with level sum equal to s. This 'HashMap' version of storage requires allocating additional memory, scaling with the number of nodes. It also does not take advantage of spatial

locality when all nodes with a given level sum must be fetched together and incurs overhead of accessing memory through the hash map.

Thus, we implemented 'Compact' storage for our sparse grids. This version takes advantage of the existing flat storage by sorting the nodes according to their 1) level sum, 2) level vector, and 3) index vector. Specifically, all nodes with global level sum equal to s are grouped together. Nodes with the same level vector $[l_1, ..., l_d]$ form a contiguous chunk of storage. Then, within this chunk the nodes are sorted using a radix-like scheme, taking advantage of the fact that level l_k of dimension k contains at most 2^{l_k-1} grid points. Thus, a node's level and index vectors can be numerically encoded into the 'id' corresponding to its index in the flat storage. Additionally, a vector `level_sum_offset` of fixed length n (another input parameter) stores the starting offset within the flat storage for the nodes with level sum s.

Implementing 'Compact' took several attempts. Firstly, the encoding strategy (computing an integer id from level and index vectors) is based on a radix sorting scheme and requires a mathematical understanding of sparse grids. We also initially populated the flat storage vector the same way as in HashMap, and then sorted according to the radix scheme, but this implementation suffered higher set-up costs and overhead, and did not extend well to the adaptive configuration. Thus, we rewrote 'Compact' to build up the storage in sorted order, required two recursive functions to enumerate the level vectors, and each level vector's index vectors.

Compared to 'HashMap' this 'Compact' storage scheme takes advantage of spatial locality when all nodes with some given level sum must be fetched, since they are all stored in a contiguous chunk of memory. Additionally, 'Compact' eliminates the layers of indirection in 'HashMap' storage, since the latter requires following pointers from the `GridNode` to its `NodeKey` to the unordered_map's underlying hash buckets to finally retrieve the id itself. Indirection induces random access patterns and inhibits the compiler's ability to prefetch, thus increasing cache misses and memory stalls. Thus, we hoped 'Compact' storage would reduce processor time spent on memory stalls and thus improve speedup.

Finally, we further optimized our sparse grid storage by omitting boundary nodes, which are at level 0 in at least one dimension and thus do not contribute at all to surplus updates of other nodes. This change reduced the amount of storage space required for both 'HashMap' and 'Compact' schemes, but also required rewrites of both storage implementations to account for this new assumption in indexing and parent/child computations.

4.3 Grid Topology

In our initial implementations, we populated the sparse grid using a "regular" topology, in which the set of grid points is determined entirely at build time by the parameters d and n. The regular grid contains all points at global levels $\mathbf{l}$ that satisfy $|\mathbf{l}|_1 + d - 1 \leq n$, and since there is no refinement, the grid never changes.

However, the Black-Scholes payoff function for a basket put has a kink along the hyperplane around the strike price: a fixed regular grid wastes grid points in smooth regions and underresolves these areas. So our fourth iteration of the solver incorporates an adaptive topology, which refines the sparse grid after the initial hierarchization by inserting additional points only where the solution is complex. This optimization is implemented via a `refine_surplus` function. After taking a snapshot of the current node count, for every active node, it inserts children in each dimension if that node's alpha value (i.e. surplus) exceeds some ϵ value. Nodes that already exist are skipped. Refinement is performed once after the initial payoff hierarchization and before stepping through iterations of the CG solver.

The Compact storage structure relies on invariants about the number and sorting of nodes by global level within the flat storage. Thus, to maintain these invariants even in an adaptive topology, we pre-populate the 1D node vector with 'inactive' dummy nodes for every new global level, then selectively activate and initialize nodes as determined by `refine_surplus`. This work-around reflects in higher node counts for Compact + Adaptive, as detailed in results below.

Compared to the regular topology, we implemented adaptive sparse grids to improve solution accuracy near the kink, at the cost of more CG iterations per timestep. This new version also increases the problem size – specifically, the number of grid nodes – and thus functions as a limit test for parallelism.

5 Results

We run various experiments to assess the accuracy, parallelism capability, and memory utilization of our Black-Scholes solver. While some hyperparameters are varied to test different problem sizes, all experiments run 50 steps of the solver and use correlated data. All input data is sourced from Heinecke et al. (see Appendix).

5.1 Computation Speedup

We measure performance via wall-clock time spent iteratively solving the PDE. In the following speedup figures, the baseline is our parallel implementation run on a single thread; thus, OpenMP task management overhead is still present. Across experiments, we vary dimension $d = 3, 4, 5, 6$ and global level limit $n = 4, 6$: these parameters control input problem size. On GHC machines, we test with $1, 2, 4, 8$ threads; on the PSC machines, we test with $1, 2, 4, 8, 16, 32, 64, 128$ threads.

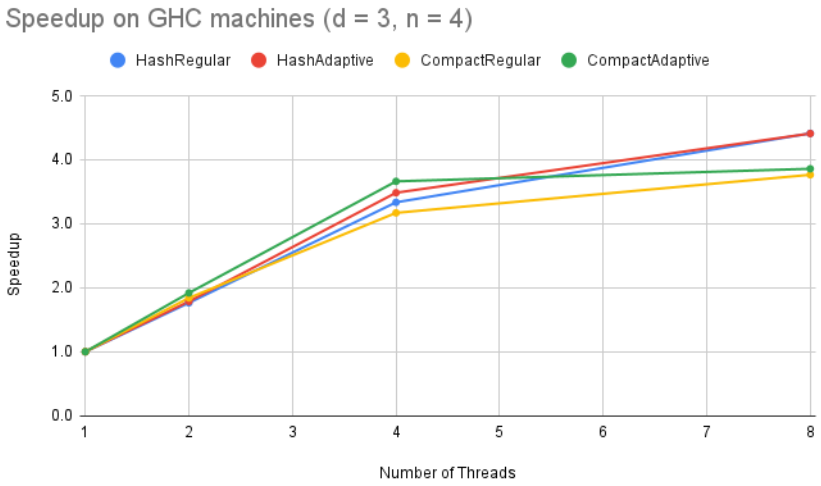

Figure 3: Computation speedup ratios (Time for 1 thread/Time for t threads) with dimension $d = 3$ and global level limit $n = 4$.

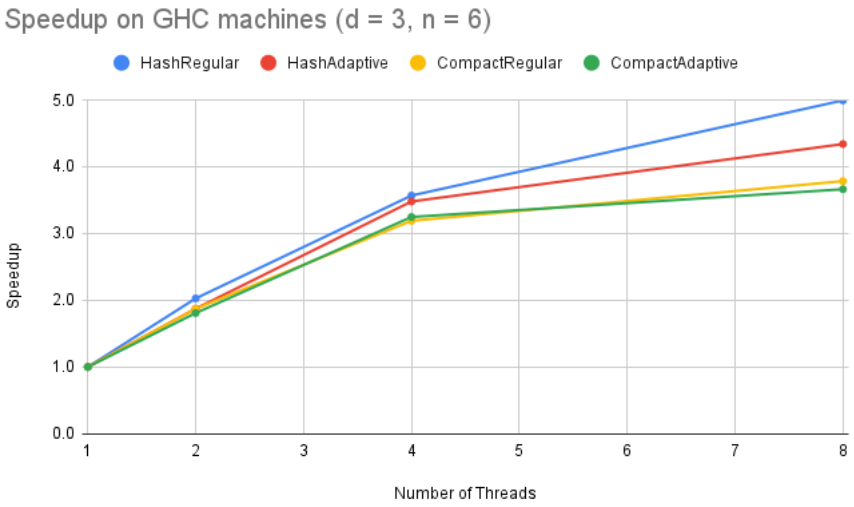

Figure 4: Computation speedup ratios (Time for 1 thread/Time for t threads) with dimension $d = 3$ and global level limit $n = 6$.

In figures 3 & 4, the number of dimensions (stocks) are fixed at 3, while the global level limit varies from 4 to 6. The speedups achieved by each version are similar in both cases, as are the trends across versions. Specifically, all four versions achieve near-linear speedup on 2 and 4 threads, but only 3.5× to 5× speedup on 8 threads. However, the magnitude of computation time scales significantly with n. For example, single-threaded Hash + Regular runtime increases from 612.1 ms to 17430.1 ms. For all four versions, increasing n from 4 to 6 increases single-threaded wall time by a factor of roughly 30×.

Thus, for subsequent experiments on varying dimension across $d = 4, 5, 6$, the global limit level is kept fixed at $n = 4$.

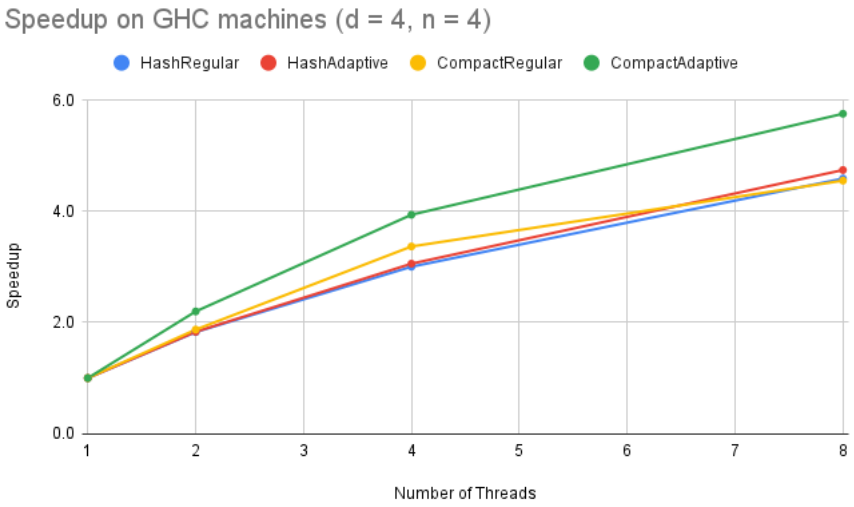

Figure 5: Computation speedup ratios (Time for 1 thread/Time for t threads) with dimension $d = 4$ and global level limit $n = 4$.

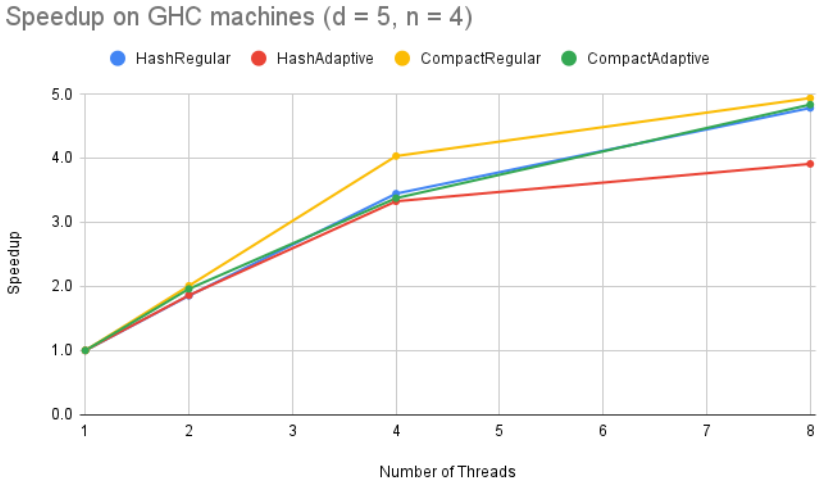

Figure 6: Computation speedup ratios (Time for 1 thread/Time for t threads) with dimension $d = 5$ and global level limit $n = 4$.

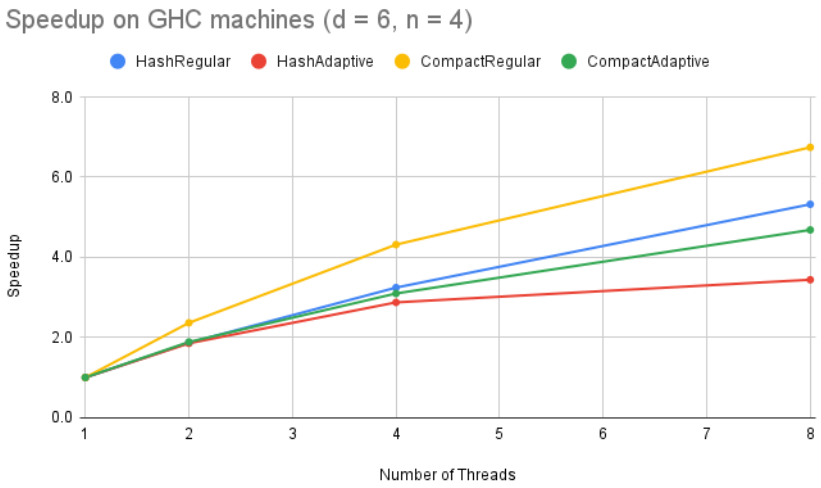

Figure 7: Computation speedup ratios (Time for 1 thread/Time for t threads) with dimension $d = 6$ and global level limit $n = 4$.

Figures 3 through 7 demonstrate how all four configurations exhibit a plateau of $4-6\times$ speedup when 8 threads are available. One source of imperfect speedup for this implementation is Amdahl's Law: as shown in the pseudocode for `updown_recursive`, the up and down sweeps for the base case of $d = 1$ execute sequentially. Additionally, once the mass and Laplacian matrices have been applied to the sparse grid, each step of the CG solver must perform some computations serially to solve the discretized equation before the next parallel step. This fraction of sequential execution does not contribute to parallelizable workload, thus preventing linear speedup.

Another source of imperfect speedup is workload imbalance. In `updown_recursive`, two independent tasks are spawned at every recursive level (ie dimension). One task performs the up sweep from finer to coarser levels within that dimension, then calls `updown_recursive` on the next dimension; the other task recurses first, then performs the down sweep on coarser to finer levels. Tasks spawned at earlier dimensions are larger than those spawned later. Additionally, the down sweep functions look at each node's two children while the up ones look at each node's one parent, so the former may also take longer. Variance in task size causes imbalance of the workload assigned to each thread. At each recursive level, the `taskwait` directive also requires all sub-tasks to complete before results can be merged. Thus, threads processing larger tasks force other threads to wait for them at the barrier. The effect of synchronization also compounds up to higher recursive levels, since each invocation of `updown_recursive` has its own barrier. We can see this manifest in the deteriorating performance of HashMap + Adaptive as d increases: adaptive topologies create more workload imbalance across tasks, since nodes are no longer distributed uniformly among levels. This makes threads more likely to accumulate idle time at barriers, and increasing d exacerbates the problem, because greater recursive depth also corresponds to more barriers.

Finally, low arithmetic intensity may also contribute to sublinear speedup. Within up- and down-sweeps, the algorithm fetches alpha values for every node at a global level, as well as their parent's or children's alphas. The sweeps also write results to the shared `b` and `c` vectors in memory. Since memory bandwidth is fixed and limited for the machine, the program becomes

bandwidth-bound when communication (memory traffic) dominates computation, preventing perfect speedup.

Within storage schemes, adaptive topologies exhibit lower speedup ratios than their regular counterparts across all experiments – with the exception of $d = 4, n = 4$ (figure 5), where Compact + Adaptive outperforms the other three versions. During adaptive refinement, children nodes (which have higher global levels) are added selectively to the sparse grid. This means only some dimensions have nodes at these new higher global levels, increasing their task size relative to other dimensions. This higher variance in task size exacerbates workload imbalance, thus reducing speedup even further below linear compared to the regular topology configurations with the same storage scheme.

Figures 3, 5, 6 and 7 demonstrate how increasing problem size via the dimension parameter improves speedup ratios. For example, HashMap + Regular and Compact + Regular both achieve their best 8-thread speedups of 5.3× and 6.7× respectively when $d = 6$ (figure 7). Recall that `apply_matvec_parallel` spawns $d + 1$ tasks, one per up-down sweep, and each of these top-level calls to `updown_recursive` spawns $\sim 2^d$ tasks via recursive calls. In fact, d controls the recursive depth of up-down. So as d increases, so does the program's parallel slack, which means OpenMP's dynamic scheduling achieves better workload balance and thus greater processor utilization even at high thread counts. This is reflected in higher speedup ratios.

Increasing the d parameter also makes Compact versions outperform their HashMap counterparts. Aside from the 6.7× vs 5.3× speedup, Compact + Adaptive achieves 4.7× to HashMap + Adaptive's 3.4× speedup on 8 threads at $d = 6$. This trend is also present at $d = 5$ (figure 6). Increasing the dimension parameter also increases the total node count, and thus the amount of memory traffic per thread. For these larger inputs, the cost of pointer-indirection, random memory access, and cache misses become more pronounced for the HashMap storage implementation, and this time spent waiting on memory stalls rather than computation impedes HashMap's speedup ratios.

In contrast, increasing problem size via the global level limit n does not consistently improve speedup ratios (3 vs 4). While d controls task count, n controls task size: the number of nodes with some global level (level vector sum) of s equals $\binom{s-1}{d-1}2^{s-d}$. Given the range $1 \leq s \leq n$, the number of nodes processed per task can vary widely, especially as n increases. Thus, increasing n does not improve parallel slack nor speedup, but rather increases the potential for workload imbalance.

Additionally, for $d = 5$ and $d = 6$, the Compact + Regular configuration achieves superlinear speedup on 2 and 4 threads. One possible explanation is that parallelism allows the threads' working sets to fit in the L2/L3 cache, whereas the working set spills to main memory in single-threading, thus increasing time spent on memory stalls.

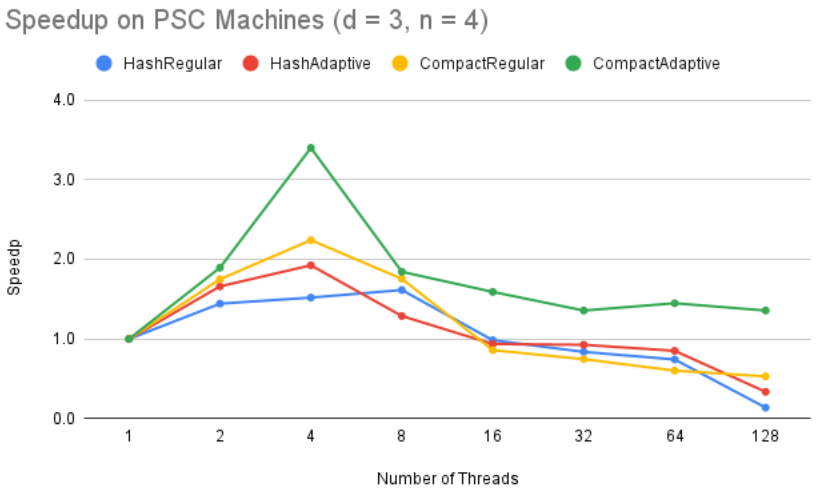

Figure 8: Computation speedup ratios (Time for 1 thread/Time for t threads) with dimension $d = 3$ and global level limit $n = 4$.

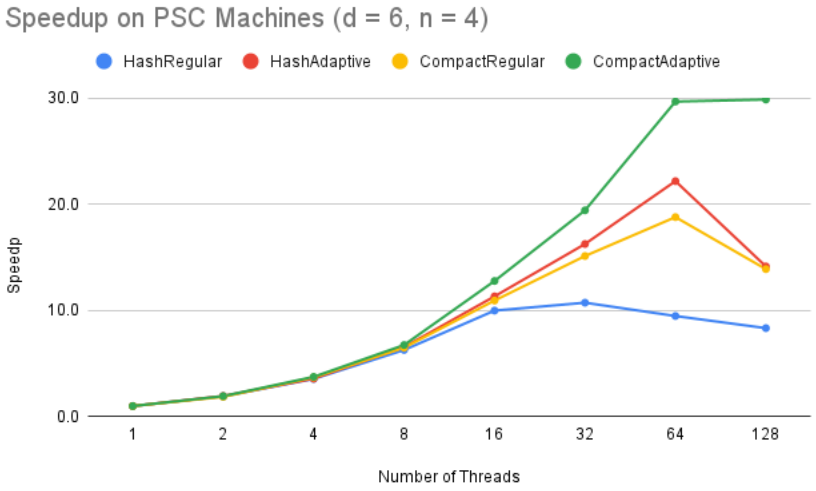

Figure 9: Computation speedup ratios (Time for 1 thread/Time for t threads) with dimension $d = 6$ and global level limit $n = 4$.

Figures 8 and 9 depict speedup ratios for smallest and largest problem sizes, when varying dimension d, on a larger range of thread counts. When $d = 3$, most configurations' speedups drop below linear as early as 4 threads and even fall below $1\times$ (longer run time than single-threading) at thread counts ≥ 16. At $d = 3$, each iteration spawns $d+1 = 4$ up-down passes, each producing $2^d = 8$ tasks. So thread count outstrips the units of parallelizable work, meaning most threads are actually idle either most or all of the time, preventing linear speedup. OpenMP's thread management overhead scaling with thread count then drives speedup below 1.

However, when $d = 6$, speedup ratios continue to improve as thread count increases to 64 (or 32 for HashMap + Regular), and stays closer to linear until 16 threads for most configurations. In these experiments, each iteration has $d + 1 = 7$ up-down sweeps spawning roughly $2^d = 64$ tasks each, producing enough parallel slack for good workload balance across even higher thread counts. However, speedup are still sub-linear, since the fraction of sequential execution (up-down base case, residual and solution update computations between parallel steps) persists, and higher thread counts also means more idle threads during this portion.

As in the speedup results on the GHC machines, Compact configurations outperform their HashMap counterparts, for the same reasons as mentioned above. However, unlike the GHC results, figure 9 demonstrates that Adaptive topology configurations outperform Regular ones when $d = 6$. While we were unable to locate memory bandwidth information about the GHC machines, we know the PSC machines' network has bandwidth 200 Gb/s. Thus, we suspect that PSC machines having a higher memory bandwidth makes them better suited to the adaptive topologies, since they induce larger task sizes (more node-dense global levels) which require more memory traffic. While bandwidth constraints cause memory stalls and thus limit speedup on the GHC machines, this bottleneck may be less of an issue on the PSC machines. When $d = 3$ (figure 8), the speedup trends for most of the configurations are not significantly different; however, Compact + Adaptive still outperforms the others at all thread counts.

5.2 Accuracy

Table 1: Metadata for each version of parallel Black Scholes solver on GHC machines with num threads = 1, $d = 3$, $n = 4$.

Version	Price	Time (ms)	Nodes	CG Iterations	Avg Iter/step
HashMap + Regular	0.051168	612.1	111	1714	34.3
Compact + Regular	0.051168	328.8	111	1714	34.3
HashMap + Adaptive	0.051189	2483.3	253	2850	57.0
Compact + Adaptive	0.051189	1683.6	351	2850	57.0

Table 2: Metadata for each version of parallel Black Scholes solver on GHC machines with num threads = 1, $d = 6$, $n = 4$.

Version	Price	Time (ms)	Nodes	CG Iterations	Avg Iter/step
HashMap + Regular	0.040259	51367.8	545	1681	33.6
Compact + Regular	0.040259	49592.8	545	1681	33.6
HashMap + Adaptive	0.040417	365981.6	2248	2423	48.5
Compact + Adaptive	0.040417	492703.8	2561	2423	48.5

We also assess the accuracy of our parallel Black-Scholes solver with respect to the reference paper. Recall degrees of freedom (dofs) translate to number of nodes. For European basket put options, Heinecke et al's solver prices four-assets at 0.058858 (397 dofs), five-assets at 0.033774 (1754 dofs), and six-assets at 0.048835 (559 dofs). Our adaptive solver prices four-assets at 0.05518 (599 dofs), five-assets at 0.027546 (1193 dofs), and six-assets at 0.040417 (2248 dofs). Thus, within some margin of error in the range 10^{-2} to 10^{-3}, our numerical results are similar to the benchmark's.

Tables 1 and 2 present metadata for single-threaded runs on $d = 3$ and $d = 6$, the smallest and largest problem sizes. In both cases, output price and number of CG iterations (and thus average iterations per step) are consistent for a given topology across both grid storage versions. This verifies that the mathematical accuracy of our solver is not impacted by the sparse grid implementation. Additionally, when number of threads is one, runtime within a topology is fairly consistent but scales drastically from regular to adaptive topology. For $d = 6$ in particular, number of nodes also increases significantly. Compact + Adaptive has slightly higher node counts than HashMap + Adaptive because of its dummy node requirement, in order to maintain invariants for the indexing math.

We also observe that the Adaptive configurations' output prices are closer to the reference values, though the absolute difference between Regular vs Adaptive report prices is small. We speculate that running more than one round of surplus refinement would increase the accuracy of adaptive grid configurations even further. However, this refinement stage is inherently sequential and occurs entirely before the iterative CG solving. So we didn't focus on improving the accuracy of adaptive refinement, since it wasn't relevant to the parallelization of our Black-Scholes solver. A single refinement pass resulted in enough irregularity in the sparse grid to produce differential speedup results.

5.3 Cache Misses

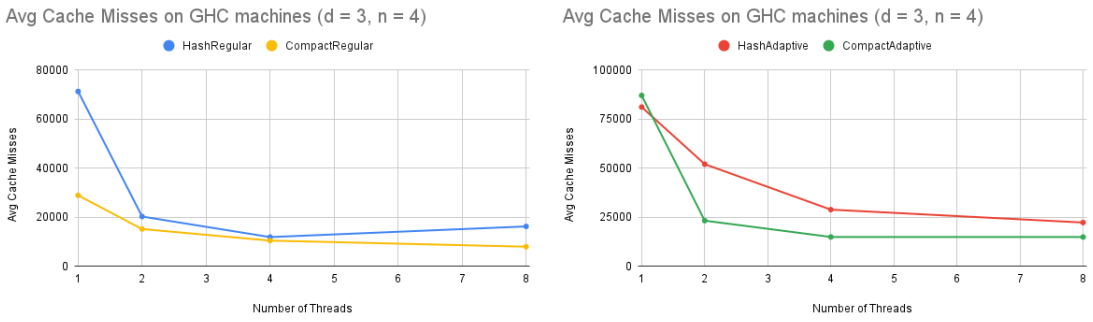

Figure 10: Average cache misses across threads, with $d = 3$ and $n = 4$.

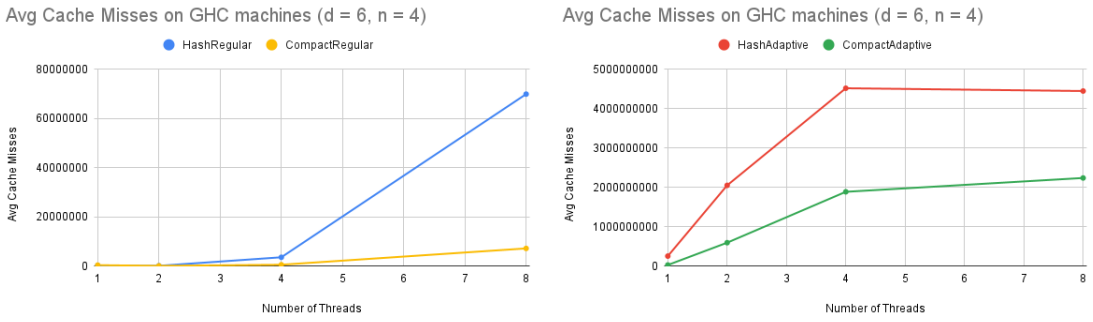

Figure 11: Average cache misses across threads, with $d = 6$ and $n = 4$.

We used `perf` to examine cache miss rate on the smallest ($d = 3, n = 4$) and largest ($d = 6, n = 4$) inputs. All four subplots show that Compact versions (almost always) produce fewer cache misses than their HashMap counterparts, across thread counts and input parameters. These results verify our motivations for the Compact storage optimization. Sorting nodes by their global level in the flat storage means a thread's working set (ie all nodes at a particular global level) is stored contiguously in memory and thus more likely to fit in fewer cache lines. This is verified by the lower average cache miss rate for Compact sparse grids. Additionally, HashMap storage suffers two layers of pointer indirection: from a GridNode to its NodeKey, then through the hash-map's internal buckets to retrieve the integer id. This random access pattern prevents the compiler from pre-fetching future accesses and causes frequent cache line evictions whenever a level sweep visits nodes that are scattered across memory.

For both $d = 3$ and $d = 6$, adaptive topologies experience cache misses at larger scales: for example, the y-axes in figure 11 demonstrate that average cache miss range is within 80,000,000 for Regular and 5,000,000,000 for Adaptive. This follows from adaptive topologies requiring much higher node counts (for example, in table 2), which span larger swatches of memory. This means some thread' working sets are less likely to fit in their caches.

Figure 10 demonstrates how on smaller inputs ($d = 3$), average cache miss rate decreases as thread count increases. Since the problem is already small enough, splitting workload over threads means each thread's working set can fit in its private L1 cache. In contrast, average cache misses increase with number of threads when the input is large ($d = 6$). Despite splitting workload, each thread's working set remains large enough to spill to main memory.

5.4 Function Profiling

TAU function profiling was performed for all versions of the solver, with the following parameters: $d = 6$, $n = 4$, and number of threads = 8.

Table 3: Mean per-rank TAU function profile (sample-based) of HashMap + Regular.

Function	**# Call**	**Incl (ms)**	**% Time**
`.TAU application`	1	10,310	100.0%
`libc (unresolved)`	506.875	2,540	24.6%
`apply_matvec_parallel`	381.125	1,907	18.5%
`libgomp (OpenMP)`	327.75	1,637	15.9%
`unordered_map lookup`	197.75	990	9.6%
`vector insert realloc`	185.375	925	9.0%
`Eigen::computeFromTridiagonal`	98.875	496	4.8%
`GridBase destructor`	67.75	339	3.3%
`enumerateIndices`	50.125	250	2.4%
`enumerateLevels`	30.875	154	1.5%

Table 4: Mean per-rank TAU function profile (sample-based) of HashMap + Adaptive.

Function	# Call	Incl (ms)	% Time
.TAU application	1	110,437	100.0%
libc (unresolved)	4805.75	24,102	21.8%
apply_matvec_parallel	4533.75	22,717	20.6%
enumerateIndices	3133.25	15,692	14.2%
libgomp (OpenMP)	2791.88	14,084	12.8%
GridBase destructor	2085	10,432	9.4%
Eigen::computeFromTridiagonal	1024.38	5,133	4.6%
Eigen::matrix_vector_product	926.875	4,651	4.2%
buildGrid	679.75	3,409	3.1%
Eigen::queryCacheSizes	439.25	2,200	2.0%

Table 5: Mean per-rank TAU function profile (sample-based) of Compact + Regular.

Function	# Call	Incl (ms)	% Time
.TAU application	1	9,032	100.0%
libgomp (OpenMP)	460.375	2,322	25.7%
Eigen::makeHouseholder	243	1,222	13.5%
up_grad	140.375	703	7.8%
libc (unresolved)	140.625	702	7.8%
Eigen::matrix_vector_product	127.125	635	7.0%
levelRank	118.5	595	6.6%
hierarchize	86.75	437	4.8%
vector insert realloc	39.125	198	2.2%
up_mass	30.625	153	1.7%

Table 6: Mean per-rank TAU function profile (sample-based) of Compact + Adaptive.

Function	# Call	Incl (ms)	% Time
.TAU application	1	106,469	100.0%
Eigen::makeHouseholder	3148.25	15,758	14.8%
libgomp (OpenMP)	3117.12	15,657	14.7%
up_mass	2351.12	11,756	11.0%
libc (unresolved)	1760.25	8,830	8.3%
Eigen::matrix_vector_product	1667.88	8,348	7.8%
Eigen::product_triangular_matrix	1596	7,991	7.5%
levelRank	1462.38	7,322	6.9%
vector insert realloc	1163	5,819	5.5%
up_grad	1141.25	5,719	5.4%

In the HashMap + Regular configuration, unordered_map lookup and vector insert reallocation consume 9.6% and 9.0% of total runtime, respectively (Table 3). Nearly 20% of execution is spent on data-structure overhead and indirection layers for accessing node data. In contrast,

the combined time spent on `levelRank` and vector insert reallocation is 8.8% for Compact + Regular (Table 5) and 12.4% for Compact + Adaptive (Table 6). Thus Compact storage optimizes accessing node data.

Additionally, storage overhead functions – including `buildGrid`, `GridBase` destructor, `enumerateIndices`, and `enumerateLevels` – contribute to 10.2% of HashMap + Regular's runtime and 26.7% of HashMap + Adaptive's runtime. On the other hand, none of these functions appear in the top 10 for either Compact configurations. These results demonstrate how Compact avoids storage overhead costs, as well. In particular, `enumerateIndices` alone contributes 14.2% to HashMap + Adaptive's runtime (its third highest); adaptive refinement substantially increases node count, especially for $d = 6$, but HashMap storage suffers the set-up cost more than Compact does.

The main parallelized function `apply_matvec_parallel` contributes the second most of execution time for both HashMap configurations, confirming it as the dominant algorithmic cost and justifying our focus on parallelizing the up-down scheme within it. On the other hand, both Compact configurations' function profiles are dominated by OpenMP runtime overhead and `Eigen` library functions. The former suggests that Compact's overhead costs come from task scheduling and synchronization, rather than storage and memory. The latter (`Eigen`) corresponds to serial execution, verifying that inherently sequential portions of the code dominate execution time.

6 Conclusion

Two serial bottlenecks are preserved in this design. First, the d=1 base case of `updown_recursive` executes the up and down sweeps sequentially, since no further branching is possible at a single dimension. Second, between each CG iteration, the solver must compute the residual, step size, and solution update serially before the next parallel matrix-vector multiplication can begin. The TAU profiles in Tables 5 and 6 corroborate this: for both Compact configurations, Eigen library functions collectively account for over 20% of total runtime, corresponding precisely to these serial CG bookkeeping steps.

Overall, we successfully parallelized a Black-Scholes solver on adaptive sparse grids. We were able to achieve decent speedup, found a relationship between speedup and problem size (i.e. increasing problem size increases speedup), and then optimizing a storage structure can help speedup via cache misses. We were also successfully able to find a different storage scheme to improve speedup as well as implement adaptive refinement for a greater accuracy in our price results.

Real basket options in practice typically involve between 3 and 30 underlying assets. Our solver targets the lower end of this range ($d = 3$ to 6), which already represents a range where full Cartesian grids are computationally impractical and sparse grid methods are useful.

A practical advantage of our approach over the industry-standard Monte Carlo method is determinism. Monte Carlo pricing introduces statistical noise whose variance decreases as $O(1/\sqrt{N})$ per sample, meaning two runs of the same pricer can return slightly different values. Our solver, by contrast, produces a single reproducible price whose error is controlled entirely by the grid resolution parameter n. This property is particularly valuable in risk management systems, where reproducible prices are often a requirement.

Sparse grids achieve faster convergence than Monte Carlo for smooth problems at low-to-moderate d, but their node count still grows super-linearly with d, whereas Monte Carlo's

$O(N)$ sampling cost is dimension-independent. This is why Monte Carlo dominates in practice for $d \geq 10$ despite its slower per-sample convergence rate. Our profiling identifies memory bandwidth as the primary bottleneck at large d since each up-down sweep requires fetching node values and their parents or children across an increasingly large working set that spills from cache to main memory. Scaling to $d \geq 10$ would therefore require distributed-memory parallelism, partitioning the node array across machines with physically separate memory buses to eliminate this bottleneck. If we had more time, trying this on a distributed machine would have been an interesting next step.

7 References

1. Heinecke, Schraufstetter, and Bungartz. "A highly parallel Black-Scholes solver based on adaptive sparse grids." *Int. J. Computer Mathematics*, 2012. [link]
2. Pflüger. "Spatially Adaptive Sparse Grids for High-Dimensional Problems." PhD thesis, TU Munich, 2010. [link]
3. Schraufstetter and Bungartz. "Parallelizing a Black-Scholes solver based on finite elements and sparse grids." *Parallel Processing and Applied Mathematics (PPAM)*, 2009. [link]

8 Work Distribution

We both contributed equally to the project. We pair-programmed the mathematical foundations, including the PAT transformation, FEM discretization, and Up-Down operator weights, which was the most time-intensive phase of development. We discussed the optimizations together and pair-programmed them as well. The writeup was split evenly.

9 Appendix

All experiments use the following parameters: strike $K = 1$, maturity $T = 1$, risk-free rate $r = 0.05$, and 50 time steps. Input parameters σ, μ, and ρ are sourced directly from Heinecke et al. [1] and are reproduced below.

Three underlyings ($d = 3$)
$\mu_1 = 0.1,\ \mu_2 = 0.02,\ \mu_3 = 0.04$
$\sigma_1 = 0.2,\ \sigma_2 = 0.3,\ \sigma_3 = 0.4$
$\rho_{1,2} = -0.7,\ \rho_{1,3} = -0.1,\ \rho_{2,3} = 0.1$

Four underlyings ($d = 4$)
$\mu_1 = \mu_2 = \mu_3 = \mu_4 = 0.05$
$\sigma_1 = 0.4,\ \sigma_2 = 0.25,\ \sigma_3 = 0.3,\ \sigma_4 = 0.4$
$\rho_{1,2} = 0.1,\ \rho_{1,3} = -0.4,\ \rho_{1,4} = 0.2,\ \rho_{2,3} = 0.3,\ \rho_{2,4} = -0.1,\ \rho_{3,4} = 0.0$

Five underlyings ($d = 5$)
$\mu_1 = \mu_2 = \mu_3 = \mu_4 = \mu_5 = 0.05$
$\sigma_1 = 0.4,\ \sigma_2 = 0.25,\ \sigma_3 = 0.3,\ \sigma_4 = 0.4,\ \sigma_5 = 0.35$
$\rho_{1,2} = 0.1,\ \rho_{1,3} = -0.4,\ \rho_{1,4} = 0.2,\ \rho_{1,5} = 0.1,\ \rho_{2,3} = 0.3,\ \rho_{2,4} = -0.1,\ \rho_{2,5} = 0.0,\ \rho_{3,4} = 0.0,\ \rho_{3,5} = 0.2,\ \rho_{4,5} = -0.7$

Six underlyings ($d = 6$)
$\mu_1 = \mu_2 = \mu_3 = \mu_4 = \mu_5 = \mu_6 = 0.05$

$\sigma_1 = 0.4,\ \sigma_2 = 0.25,\ \sigma_3 = 0.3,\ \sigma_4 = 0.4,\ \sigma_5 = 0.35,\ \sigma_6 = 0.4$
$\rho_{1,2} = 0.1,\ \rho_{1,3} = -0.4,\ \rho_{1,4} = 0.2,\ \rho_{1,5} = 0.1,\ \rho_{1,6} = 0.3,\ \rho_{2,3} = 0.3,\ \rho_{2,4} = -0.1,\ \rho_{2,5} = 0.0,\ \rho_{2,6} = 0.0,\ \rho_{3,4} = 0.0,\ \rho_{3,5} = 0.2,\ \rho_{3,6} = 0.1,\ \rho_{4,5} = -0.7,\ \rho_{4,6} = -0.3,\ \rho_{5,6} = 0.5$

Raw Execution Times + Cache Miss Data

Table 7: Wall Time (seconds) and Computational Speedup ($d = 3$, $n = 4$, GHC)

	HashRegular		**HashAdaptive**	
Threads	**Wall Time**	**Speedup**	**Wall Time**	**Speedup**
1	612.1	1.0	2483.3	1.0
2	347.0	1.8	1390.2	1.8
4	183.5	3.3	712.5	3.5
8	138.7	4.4	563.0	4.4
	CompactRegular		**CompactAdaptive**	
Threads	**Wall Time**	**Speedup**	**Wall Time**	**Speedup**
1	328.8	1.0	1683.6	1.0
2	178.6	1.8	877.4	1.9
4	103.7	3.2	459.6	3.7
8	87.3	3.8	436.2	3.9

Table 8: Wall Time (seconds) and Computational Speedup ($d = 3$, $n = 6$, GHC)

	HashRegular		**HashAdaptive**	
Threads	**Wall Time**	**Speedup**	**Wall Time**	**Speedup**
1	17430.1	1.0	46859.9	1.0
2	8598.2	2.0	25018.6	1.9
4	4884.5	3.6	13467.6	3.5
8	3490.4	5.0	10797.4	4.3
	CompactRegular		**CompactAdaptive**	
Threads	**Wall Time**	**Speedup**	**Wall Time**	**Speedup**
1	10038.6	1.0	37342.9	1.0
2	5363.5	1.9	20659.3	1.8
4	3147.1	3.2	11500.2	3.2
8	2652.6	3.8	10195.6	3.7

Table 9: Wall Time (seconds) and Computational Speedup ($d = 4$, $n = 4$, GHC)

	HashRegular		**HashAdaptive**	
Threads	**Wall Time**	**Speedup**	**Wall Time**	**Speedup**
1	2580.5	1.0	11110.7	1.0
2	1411.9	1.8	6063.4	1.8
4	859.5	3.0	3634.8	3.1
8	562.5	4.6	2343.5	4.7
	CompactRegular		**CompactAdaptive**	
Threads	**Wall Time**	**Speedup**	**Wall Time**	**Speedup**
1	1502.4	1.0	11194.2	1.0
2	802.7	1.9	5092.2	2.2
4	446.3	3.4	2843.6	3.9
8	330.2	4.5	1945.9	5.8

Table 10: Wall Time (seconds) and Computational Speedup ($d = 5$, $n = 4$, GHC)

	HashRegular		**HashAdaptive**	
Threads	**Wall Time**	**Speedup**	**Wall Time**	**Speedup**
1	13491.6	1.0	74467.7	1.0
2	7279.1	1.9	40035.4	1.9
4	3915.1	3.4	22382.8	3.3
8	2820.7	4.8	19043.3	3.9
	CompactRegular		**CompactAdaptive**	
Threads	**Wall Time**	**Speedup**	**Wall Time**	**Speedup**
1	10005.2	1.0	95624.8	1.0
2	4981.4	2.0	48790.2	2.0
4	2481.1	4.0	28328.2	3.4
8	2026.6	4.9	19775.5	4.8

Table 11: Wall Time (seconds) and Computational Speedup ($d = 6$, $n = 4$, GHC)

	HashRegular		**HashAdaptive**	
Threads	**Wall Time**	**Speedup**	**Wall Time**	**Speedup**
1	51367.8	1.0	365981.6	1.0
2	27537.1	1.9	197569.4	1.9
4	15836.0	3.2	127313.9	2.9
8	9649.9	5.3	106403.0	3.4
	CompactRegular		**CompactAdaptive**	
Threads	**Wall Time**	**Speedup**	**Wall Time**	**Speedup**
1	49592.8	1.0	492703.8	1.0
2	20975.7	2.4	261273.6	1.9
4	11489.5	4.3	159093.0	3.1
8	7352.1	6.7	105207.3	4.7

Table 12: Total and Average Cache Misses ($d = 3$, $n = 4$, GHC)

	HashRegular		**HashAdaptive**	
Threads	**Total**	**Avg**	**Total**	**Avg**
1	71,334	71,334	81,204	81,204
2	40,525	20,263	104,123	52,062
4	47,521	11,880	115,435	28,859
8	129,825	16,228	178,206	22,276
	CompactRegular		**CompactAdaptive**	
Threads	**Total**	**Avg**	**Total**	**Avg**
1	28,923	28,923	87,136	87,136
2	30,340	15,170	46,471	23,236
4	41,788	10,447	59,650	14,913
8	63,640	7,955	119,104	14,888

Table 13: Total and Average Cache Misses ($d = 6$, $n = 4$, GHC)

	HashRegular		**HashAdaptive**	
Threads	**Total**	**Avg**	**Total**	**Avg**
1	173,746	173,746	251,708,573	251,708,573
2	157,668	78,834	4,102,216,578	2,051,108,289
4	14,202,611	3,550,653	18,086,187,212	4,521,546,803
8	559,638,536	69,954,817	35,604,878,817	4,450,609,852
	CompactRegular		**CompactAdaptive**	
Threads	**Total**	**Avg**	**Total**	**Avg**
1	354,171	354,171	25,576,282	25,576,282
2	135,106	67,553	1,185,796,894	592,898,447
4	2,072,993	518,248	7,548,807,357	1,887,201,839
8	57,227,285	7,153,411	17,912,622,500	2,239,077,813

Table 14: Wall Time (seconds) and Computational Speedup ($d = 3$, $n = 4$, PSC)

	HashRegular		**HashAdaptive**	
Threads	**Wall Time**	**Speedup**	**Wall Time**	**Speedup**
1	707.6	1.0	2650.3	1.0
2	490.4	1.4	1597.0	1.7
4	465.8	1.5	1377.4	1.9
8	438.4	1.6	2058.2	1.3
16	719.0	1.0	2830.7	0.9
32	844.7	0.8	2859.4	0.9
64	952.6	0.7	3113.5	0.9
128	5106.9	0.1	7867.6	0.3
	CompactRegular		**CompactAdaptive**	
Threads	**Wall Time**	**Speedup**	**Wall Time**	**Speedup**
1	566.5	1.0	3475.9	1.0
2	323.9	1.7	1835.6	1.9
4	252.9	2.2	1023.1	3.4
8	322.8	1.8	1886.5	1.8
16	659.2	0.9	2185.5	1.6
32	758.9	0.7	2562.2	1.4
64	939.7	0.6	2400.4	1.4
128	1068.3	0.5	2561.4	1.4

Table 15: Wall Time (seconds) and Computational Speedup ($d = 6$, $n = 4$, PSC)

	HashRegular		**HashAdaptive**	
Threads	**Wall Time**	**Speedup**	**Wall Time**	**Speedup**
1	62766.0	1.0	423812.8	1.0
2	32901.7	1.9	218140.8	1.9
4	17664.7	3.6	118435.7	3.6
8	10002.0	6.3	64684.7	6.6
16	6288.6	10.0	37434.8	11.3
32	5853.1	10.7	26085.4	16.2
64	6626.6	9.5	19123.5	22.2
128	7533.4	8.3	29978.0	14.1
	CompactRegular		**CompactAdaptive**	
Threads	**Wall Time**	**Speedup**	**Wall Time**	**Speedup**
1	108660.2	1.0	1557514.1	1.0
2	58208.2	1.9	805348.2	1.9
4	29422.5	3.7	415558.0	3.7
8	16733.6	6.5	230934.0	6.7
16	9944.2	10.9	121987.4	12.8
32	7188.3	15.1	80220.8	19.4
64	5786.8	18.8	52531.0	29.6
128	7826.6	13.9	52197.4	29.8