Parallel Black-Scholes Solver on Adaptive Sparse Grids

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

Updated Schedule:

Week	Tasks
Week 1(Mar 25 – Apr 1)	Read the Pfluger thesis (Ch. 2–4) and Heinecke et al. to understand the data structure and algorithm. Build the sparse grid in C++: store nodes by level and index, implement neighbor lookups, handle boundaries.
Week 2(Apr 2 – Apr 8)	Implement the serial and initial OpenMP parallel versions of Up-Down algorithm and associated methods like hierarchization.
Week 3(Apr 9 – Apr 14)	**Milestone.** Run correctness unit tests and compare static and dynamic scheduling for Up-Down code.
Week 4(Apr 15 – Apr 22)	Apr 15 – Apr 18: Implement time-stepping and incorporate all mathematical constraints. Check 2D and 3D option prices match known values. Apr 19 – Apr 22: Optimize and iterate on parallel implementation. Compact sparse grid representation. Run further experiments.
Week 5(Apr 23 – Apr 30)	Apr 23 – 26: Add adaptive refinement near the strike price. Run experiments to compare with uniform implementation. Apr 27 – 30: Finish all experiments and plots. Write up final report and produce poster. Stretch goals: attempt Heston stochastic volatility and/or American option early exercise.

Completed Work + Results:

The bulk of our effort and work so far has focused on understanding and implementing the following mathematical pipeline behind an adaptive sparse grid-based Black-Scholes solver.

- Principal Axis Transformation: apply PAT to transform the Black-Scholes PDE to the simpler multi-dimensional heat equation.
- Domain Sizing: this heat equation lives in R^d, but a finite domain is required to solve this problem numerically, so we transform to $[0, 1]^d$.
- ✓ Sparse Grid: this component makes the problem mathematically feasible by avoiding the "curse of dimensionality." Each node in the grid is identified by a level vector and an index vector, and some node (l, i) is included only if the global level sum $l_k \leq n + d - 1$. Nodes are grouped by global level.
- ✓ Set Inital conditions and Hierarchization: We convert raw numbers into the coarser and finer grain levels by setting nodal values to "hierarchical surpluses." Level 1 nodes contain

their actual node values, while nodes at higher (finer) levels subtract from the coarser predictions.

- ✓ Up-Down sweep: reduce the d-dimensional formulation to simple 1-d operations by reusing results calculated in other dimensions. Each one-dimensional operation (ie scalar product) is split into two passes: up for accumulating information from finer levels and down for accumulating information from coarser levels.
- Laplacian: second sweep over the sparse grid to calculate the second derivatives, in order to compute the PDE.
- Crank Nicolson Time stepping: Iterate over time steps using a mass matrix and the Laplacian operator. Use Rannacher smoothing, which applies a damping time-step method at earlier steps to deal with non-smooth functions, before switching to Crank Nicolson.
- Conjugate gradient (CG) solver: rather than explicitly constructing matrices, at each time step computes the matrix-vector products by applying the Up-Down and Laplacian methods to the sparse grid.

Summary

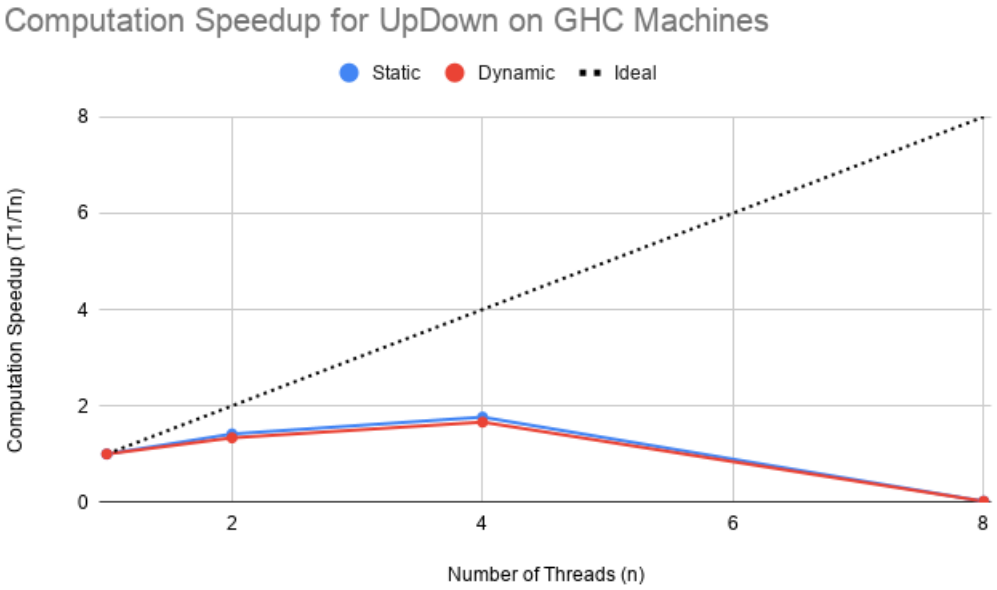
After breaking down the finite-element Black-Scholes solver into this pipeline, we implemented the core data structures and algorithms in C++. First, we built components of the Sparse Grid representation, including hashable `NodeKey`s encoding level and index, `GridNode`s containing both nodal values and surpluses (called alphas), and a `SparseGrid` class with an underlying unordered map. This class also implements helper functions to evaluate a node at each hierarchical piecewise linear basis and to interpolate a given function onto the grid.

We also implemented the core Up-Down algorithm from Heinecke et al, which performs two passes over the sparse grid: one passing information from finer to coarser levels (up) and the other from coarser to finer (down). All nodes at the same level are independent and can be processed simultaneously, but level l cannot begin until level $l+1$ is complete during Up or until level $l-1$ completes during Down. Thus, we parallelize within a level but use a synchronization barrier between levels. We implement this algorithm under the shared address space model, using OpenMP, and experiment with both static and dynamic task scheduling of nodes to threads within a level. We also implemented a parallel version of hierarchization, to compute the hierarchical surpluses at each node, and a sequential version of de-hierarchization, to convert them back into values.

Results

To verify and analyze our implementation so far, we performed correctness unit tests and speedup experiments. For Sparse Grid, we tested the grid generation, basis evaluation, and 1-D & 2-D interpolation functions and verified the correctness of each. For Up-Down and Hierarchization, we first ensured correctness of our parallel implementations by comparing the computed alpha values (from hierarchize) and the results of matrix-vector multiplication (from up-down) across thread values of $n = 1, 2, 4, 8$. In comparison to single-threaded execution, tests on other values of n produced consistent results within a tolerance level of 10^{-12}.

We also analyzed the computation speedup of this preliminary parallel Up-Down implementation, using both static and dynamic scheduling of tasks (ie nodes) within levels.



While $n = 2, 4$ result in some sub-linear speedup (less than a factor of 2 in both cases), at $n = 8$ the ratio $\frac{T_1}{T_n}$ falls to less than 0.2, indicating that execution takes 5 times longer. There are two possible reasons for these results. The first is problem size: as preliminary tests were run with smaller values for dimension and maximum level of the sparse grid, it is possible that overhead dominates increased parallelism at high n. In particular, if most levels contain $< n$ nodes, then threads are idle more often than not. To further investigate this issue, we will source true options-pricing data and generate larger test cases, whose workload is more likely to benefit from parallelism.

Another potential reason for these results is that the problem is bandwidth-bound. For each task/node, one processor must make several memory accesses: retrieving the current key, its corresponding node in the sparse grid, the mesh width h, the up or down weight associated with h and the current level – all before updating the key's temporary result. Even as the number of threads increases, the machine's bandwidth is limited, so threads will often suffer memory stalls. One potential solution we plan to investigate is to compact the sparse grid representation. We will replace the unordered map representation with a 2D array that stores node information by level, for each dimension. This restructuring will hopefully improve cache locality and avoid the layers of indirection inherent in a hash map.

Reflection + Updated Goals:

We realized that the math is a little more complicated than anticipated. We had to parse a lot of mathematical equations and it took way longer to understand than we had expected initially. Because of this, we decided to focus on getting a working sparse grid and up-down implementation to start benchmarking while we simultaneously work on the math part of it.

Our main next steps with the math aspect is to do the PrincipalAxisTransform, and Conjugate gradient solver. The algorithms are in the paper, but we have yet to implement them. With the Eigen math library, it should be doable within the next week or so. The PrincipalAxisTransform is just computing the eigendecomposition of the covariance matrix and will reduce the PDE to

an uncorrelated heat equation. The conjugate gradient solver is calculating A $*$ x = b. This method is tailored to sparse grids so we never have to build the full matrix representation.

After implementing a fully-functioning and parallel Black-Scholes solver, we plan to focus on adaptive refinement. The payoff function has a sharp kink at the strike price K, where the derivatives become discontinuous and thus making uniform sparse grids converge more slowly at this kink. Thus, adaptivity – using more grid points near this kink and fewer where finer granularity is less necessary – improves the sparse grid's accuracy. However, the grid also becomes more irregular, making the parallelization of the algorithms more interesting. We plan to implement adaptive refinement and compare both accuracy and speedup to the uniform case.

Our only remaining concerns/unknowns relate to the mathematical complexity of the problem. Even a purely serial Black-Scholes solver requires a strong understanding of the formulas, their derivations, and how they pipeline together. Our current resources include several papers describing previous work in this field. Additionally, we plan to take advantage of existing C++ libraries when implementing the more math-focused methods and algorithms.

We also anticipate that time constraints will prevent us from exploring all of our proposed stretch goals. Thus our focus, rather than trying to extend to different parts of the financial problem, will be on the adaptive sparse grid representation and, as mentioned above, compacting the grid in different ways. If time does permit, we make investigate incorporating stochastic volatility or American 'early exercise' options.

At the poster session, we plan to use graphs and tables to display findings from our various experiments: computational speedup across thread counts, static vs dynamic task scheduling, uniform vs adaptive sparse grid implementations.