15418 25 FALL project Milestone report

Parallelizing the Longstaff–Schwartz Algorithm for American Option Pricing

A. Summary of work completed so far

Since the submission fo the project proposal, we have made substantial progress on the baseline system the the performance analysis required for the parallelization

1. A Fully sequential baseline implemented c++ 17

```
yunfanw2@ghc28:~/private/15418/longstaff/build$ ./lsm_price --S0 100 --K 100 --r 0.0
5 --sigma 0.2 --T 1.0 --paths 50000 --steps 50 --put
Running Longstaff-Schwartz baseline (sequential)
  S0    = 100
  K     = 100
  r     = 0.05
  sigma = 0.2
  T     = 1
  paths = 50000
  steps = 50
  deg   = 2
  seed  = 42
  type  = American Put

Estimated American option price: 6.04053
```

CMakeLists.txt | main.cpp | lsm.hpp | *buildTest.txt*

private > 15418 > longstaff > buildTest.txt

```
1   cmake ..
2   cmake --build .
3   --S0 100 --K 100 --r 0.05 --sigma 0.2 --T 1.0 --paths 50000 --steps 50 --put
4   Sample explanation:
5   S0 = 100    Current stock price is 100
6   K = 100 Strike price is 100
7   r = 5%  Risk-free interest rate is 5%
8   sigma = 20% Volatility of the stock is 20%
9   T = 1.0 Option matures in 1 year
10  paths = 50000   Monte Carlo uses 50k simulated stock paths
11  steps = 50  Each path has 50 time points (discrete exercise opportunities)
12  --put   We are pricing a put (not a call)
```

We now have a fully working, verified, sequential implementation of Longstaff–Schwartz (LSM) for pricing American call/put options.

The basic features of it are:
Implemented GBM path generatio, Implemented regression-based continuation value estimator, Implemented backward exercise decision, Implemented CLI and CMake build system. It has a verified correctness with stable pricing results matching known industry ranges (European vs American pricing gap)

2. Identified and Marked all parallelization Opportunities

We analyzed the code path by path and timestep by timestep.Our analysis confirms the expected high-level structure:
Outer loop over time steps must remain sequential due to optimal stopping dependency. All inner loops over paths are embarrassingly parallel, except for a very small regression solve. The regression system $(X^TX)\beta = X^TY$ decomposes into parallel reductions and we have identified all the possible places in the code to exploit parallelism:
Option pricer's parallelism:

```
// POSSIBLE PARALLELISM 1
// Core: Longstaff-Schwartz American option pricer

inline double price_american_lsm(const ModelParams& model,
                                 const OptionParams& opt,
                                 const LsmConfig& cfg)
{
    std::size_t N = cfg.num_paths;
    std::size_t M = cfg.num_steps;
    double dt = opt.T / static_cast<double>(M);
    double disc = std::exp(-model.r * dt);

    // S1. Simulate all paths
    auto paths = simulate_paths(model, opt, cfg);

    // S2. Initialize cashflows at maturity
    std::vector<double> cashflow(N, 0.0);
    std::vector<std::size_t> exercise_index(N, M); // time index of exercise
    // POSSIBLE PARALLELISM 1.1
    for (std::size_t i = 0; i < N; ++i) {
        cashflow[i] = payoff(opt, paths[i][M]);
        // if it's zero, it just means never in the money, that's fine
    }

    // S3. Backward induction over time steps M-1, ..., 1
    const std::size_t deg = cfg.poly_degree;
    const std::size_t K = deg + 1;

    std::vector<std::size_t> itm_paths;  // indices of in-the-money paths
    itm_paths.reserve(N);

    for (int j = static_cast<int>(M) - 1; j >= 1; --j) {
        itm_paths.clear();

        // Find paths that are STILL alive and in-the-money at step j
        // POSSIBLE PARALLELISM 1.2
        for (std::size_t i = 0; i < N; ++i) {
            // If exercise_index[i] <= j, it means the option has already bee
            if (exercise_index[i] <= static_cast<std::size_t>(j)) {
                continue;
            }
```

Parallelism 1.1: Calculating the cashflow in parallel with payoff function.
Parallelism 1.2: Parallel backward induction outer loop for each path.

```
        // POSSIBLE PARALLELISM 1.3
        for (std::size_t idx : itm_paths) {
            double S = paths[idx][j];
            // Discount future cashflow from exercise_index[idx] back to time
            std::size_t ex_idx = exercise_index[idx];
            double tau = static_cast<double>(ex_idx - j);
            double Y = cashflow[idx] * std::exp(-model.r * dt * tau);

            auto phi = evaluate_basis(S, deg); // length K

            // Accumulate A += phi * phi^T, b += phi * Y
            for (std::size_t r = 0; r < K; ++r) {
                for (std::size_t c = 0; c < K; ++c) {
                    A[r * K + c] += phi[r] * phi[c];
                }
                b[r] += phi[r] * Y;
            }
        }

        // Solve A beta = b
```

Parallelism 1.3: Parallel cashflow evaluation for each potential path

```
// POSSIBLE PARALLELISM 1.4
for (std::size_t idx : itm_paths) {
    double S = paths[idx][j];
    double ex_payoff = payoff(opt, S);
    if (ex_payoff <= 0.0) continue;

    auto phi = evaluate_basis(S, deg);

    double cont_est = 0.0;
    for (std::size_t k = 0; k < K; ++k) {
        cont_est += beta[k] * phi[k];
    }

    if (ex_payoff >= cont_est) {
        // Exercise now: overwrite cashflow and exercise index
        cashflow[idx] = ex_payoff;
        exercise_index[idx] = static_cast<std::size_t>(j);
    }
}
```

Parallelism 1.4: Parallel exercise decision for all the in money paths and exercise it now.

```cpp
    // POSSIBLE PARALLELISM 1.5
    double sum = 0.0;
    for (std::size_t i = 0; i < N; ++i) {
        std::size_t j = exercise_index[i];
        double t = dt * static_cast<double>(j);
        sum += cashflow[i] * std::exp(-model.r * t);
    }
```

Parallelism 1.5: Summing all the cashflow in parallel

```cpp
// POSSIBLE PARALLELISM 2
//1: simulate GBM paths at risk-neutral measure

// Returns a matrix paths[path][time_index]
// with time_index = 0to num_steps, where time 0 =S0 and time M=T.
//
inline std::vector<std::vector<double>>
simulate_paths(const ModelParams& model,
               const OptionParams& opt,
               const LsmConfig& cfg)
{
    std::size_t N = cfg.num_paths;
    std::size_t M = cfg.num_steps;
    double dt = opt.T / static_cast<double>(M);

    std::vector<std::vector<double>> paths(N, std::vector<double>(M + 1));
    std::mt19937_64 rng(cfg.rng_seed);
    std::normal_distribution<double> std_normal(0.0, 1.0);

    double drift = (model.r - 0.5 * model.sigma * model.sigma) * dt;
    double vol_sqrt_dt = model.sigma * std::sqrt(dt);

    for (std::size_t i = 0; i < N; ++i) {
        paths[i][0] = model.S0;
        for (std::size_t j = 1; j <= M; ++j) {
            double z = std_normal(rng);
            double log_growth = drift + vol_sqrt_dt * z;
            paths[i][j] = paths[i][j - 1] * std::exp(log_growth);
        }
    }

    return paths;
}
```

Parallelism 2: the path simulation using monte carlo methods's naive parallelism

3. Designed Parallel Locality-Aware Backend Architecture and further potential algorithm design
   We have designed the plan to improve spatial locality and reduce memory traffic during backward steps:

financial derivatives in a straightforward manner (such as replacing the pseudorandom

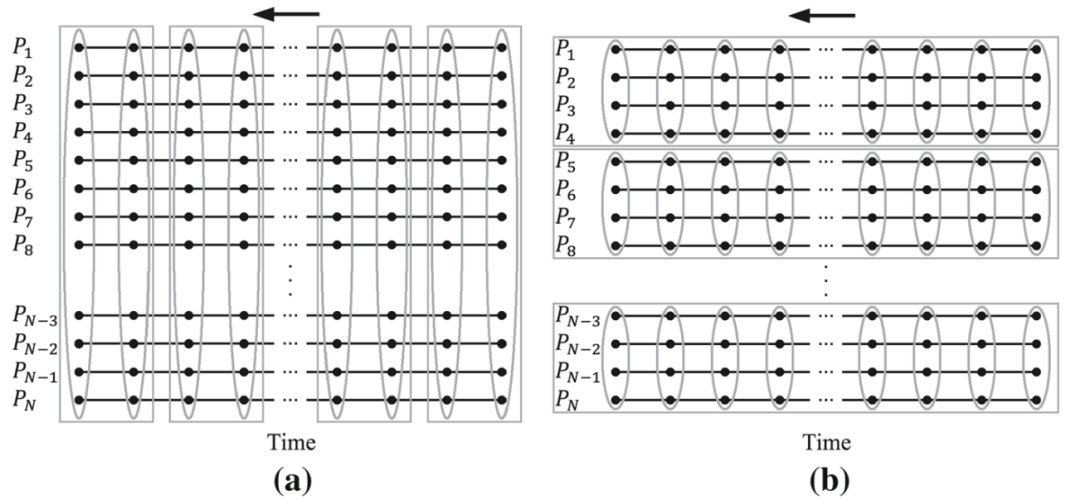
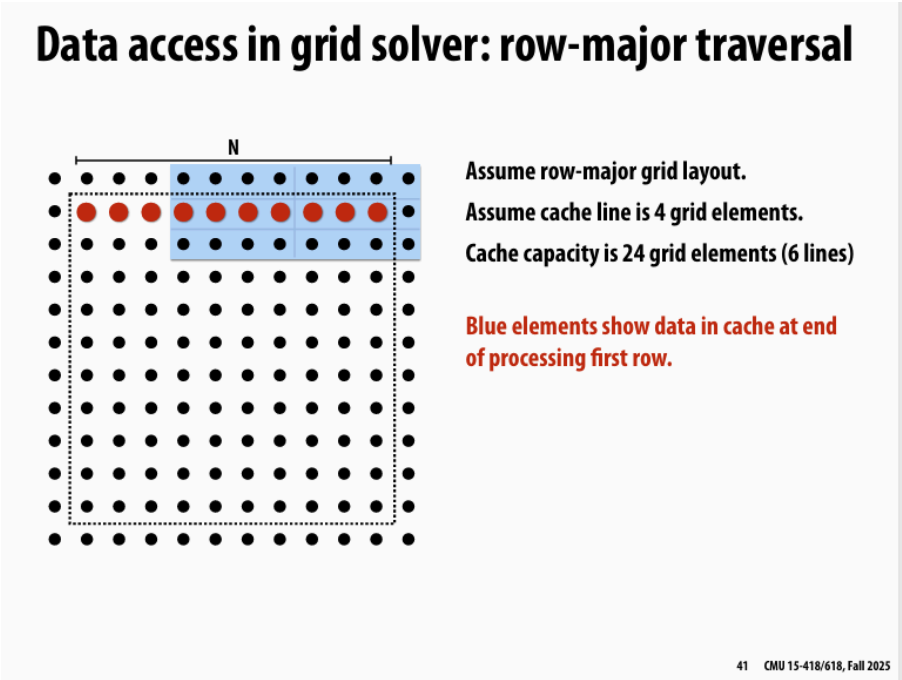


Fig. 2 Two ways to parallelize the regression phase. **a** Time decomposition. **b** Space decomposition. P_i is the ith path. A rectangle encloses the paths assigned to a process. An ellipse encloses the prices used during the regression phase within a process for a particular time step

In the paper we discussed in the proposal, we finally lean towards the model b in the above figure and we are planning to further design the parallel architecture in the grid layout as discussed in the lecture 6's slides (as is shown below). Since within each backward induction step, the standard procedure is to for each timestep path, the value each node for current step is going to use all the results from previous time step. But if we draw the destination value randomly and assign the values randomly to each path, we can prove that each value, when we only use the results from adjacent blocks, we can get the same expected value as the result we use the results from all other blocks (which is the standard procedure). This is a bootstrapping method and can greatly make use of space locality to exploit parallelism.



B. Status vs proposal goals

We are on track with Major goals:
We proposed:
Build a sequential LSM baseline, parallelize across path, parallelize regression (XTX and XTY reductions), implement CPU parallelism (OpenMP or pthreads), implement GPU parallelism (CUDA), benchmark speedups vs baseline and generate graphs.
We are now half way through the project, in the middle of implementing GPU parallelism and CPU parallelism, many critical sections are done, there are more testing and fine tuning left to do.

Some Nice to haves are:
Instead of just having naive GBM methods, we can have multi-model support to accommodate more ways to simulate the path. We can also have batch pricing support if the GPU code is stable.

Deliverables we believe we will finish for poster session:
Parallel CPU version (OpenMP)
Parallel GPU version (CUDA, path-per-thread design)
Runtime scaling graphs (scaling in N paths, speedup, efficiency curves)
Visualization of exercise boundary learned by LSM
Comparison to sequential baseline

C. What We Will Show at the Poster Session

1. Live Demo for the GPU, CPU runtime difference for pricing the same option.
2. Performance graphs
3. Summary table

There are preliminary results for correctness check, which is some approximate fine results of sequential models.

The tested sequential runtime for 100k paths, 50 time steps are 1.2-1.8.

D. Issues and unknowns

We are eager to see if our optimization will bring the performance to what level and how, maybe, instead of GPU, we can use TPU for backwards reduction, which will definitely have better accuracy, but to what level the accuracy will be? And how the result will be closer to the closed form one shot solutions? How will that harm the performance from the case of GPU?