Accelerating Financial Code through Parallelisation and Source-Level Optimisation

Nan Zhang and Ka Lok Man

Abstract—In this paper we summarise the experiences we obtained during past years in accelerating financial code through parallelisation and source-level optimisation. We have been focusing on developing optimised parallel programs to speedup financial computations where either binomial tree method or Monte Carlo simulation was applicable. The parallelisation was through explicit POSIX multi-threading on x86 shared-memory multi-processor systems. The source-level optimisations we found most useful were data structure optimisation and elimination of common sub-expressions.

Index Terms—Parallel computing, Monte Carlo simulation, Binomial tree method, Source code optimisation, POSIX multi-threading

I. INTRODUCTION

SOFTWARE routines that solve computational finance problems are often time and resource consuming. The purpose of this paper is to briefly discuss two practical approaches to accelerate such financial routines, namely, parallelisation and source code optimisation. We will discuss their applications on binomial tree method and Monte Carlo simulation. The purpose is to speedup their executions on x86 shared-memory multi-processor systems. Both these two methods are popular computational approaches in quantitative finance. Binomial tree method is often applied in pricing American-style options whose built-in feature of early exercise cannot be handled by analytical methods. However, for those complex options whose value depends on a basket of basic assets with high-dimensionality Monte Carlo simulation is often the only effective way to evaluate their price.

The parallelisation under our discussion is achieved through explicit POSIX multi-threading. In this approach one has to explicitly create and manage threads through invoking corresponding POSIX functions. Although it requires much more programming efforts than automatic multi-threading through OpenMP directives, the performance is much better. The code optimisations we are going to discuss include data structure optimisation and elimination of common sub-expressions.

II. EXPLICIT MULTI-THREADING IN BINOMIAL TREE METHOD

The binomial tree method is an often-used approach in financial computing to solve problems like option pricing. To parallelise the computation on a binomial tree the key things are how to decompose the tree so that different threads work on different parts independently, and how to synchronise the working threads. Fig. 1 shows such an example where the nodes in a 12-level binomial tree are processed by three threads in parallel. The parallel algorithm reported in [1] partitions a binomial tree into blocks of multiple levels. A block is further divided into sub-blocks. Each sub-block is assigned to a working thread. All working threads in parallel process the sub-blocks in a block, and then, when finishing, they move to the next block. The algorithm is an improved version from the one presented in [2].

A slightly modified algorithm is designed to work on CPU-GPU heterogeneous platforms [3]. Because on a GPU, accessing local memory is much more faster than accessing global memory, the GPU binomial tree algorithm uses double buffers in shared local memory to reduce the times the global memory has to be accessed.

III. EXPLICIT MULTI-THREADING IN MONTE CARLO SIMULATION

Monte Carlo simulation is another popular method used to solve complex computational finance problems. In a typical application of this approach a large number of scenarios are generated, and computations are performed on each of these scenarios. Usually, the computations performed on one scenario is independent of the computations performed in another scenario. For this reason, it is natural to parallelise Monte Carlo simulations using multiple processors. For instance, on a shared-memory multi-processor system...
if the number of processors is $p$, the number of scenarios to generated is $n$, a simple parallelisation scheme is to divide the $n$ scenarios evenly divided among the $p$ processors so that each processor works on $n/p$ scenarios.

Monte Carlo approach often involves using random numbers. It is more desirable if the generation of random numbers is also parallelised. For this purpose the random number generator must support the “skipping ahead” method, such as the ones in Intel MKL [4]. With the same definitions for numbers $n$ and $p$, if $n$ is the number of random variates needed in one scenario, for the parallel generation to work and parallelisation within the same mathematical expression there are common sub-expressions. In many cases, because of the way those sub-expressions are constructed there are common parts in them. To save computational time we can compute the common part once and save its value and re-use this value in subsequent computations. For example, at the $k$-th level of a binomial tree, stock prices represented by the nodes are $S_0 u^k, S_0 u^{k-2}, \ldots, S_0 u^{-k}$, where $S_0$ is the initial stock price and $u$ is the up-move factor. To compute these values we let $X_0 = S_0 u^k$, $X_1 = S_0 u^{k-2}$, $X_2 = S_0 u^{k-4}$, etc, and $U = u^{-2}$. We can see that $X_1 = X_0 U$, $X_2 = X_1 U$, $X_3 = X_2 U$, etc. So, in runtime what we can do is to save the value of $X_1$ and re-use it to compute $X_{n+1}$. This saves execution time because multiplication takes much less time than the transcendental operation in computing $u^k$.

This makes a big difference, especially when the option pricing routine gets called repeatedly by some higher-level procedure, as in the case of calculating implied volatilities [8]. Another example that shows such optimisation works is reported in [6], where in computing the drift term in the extended LIBOR market model this optimisation saves a big amount of execution time. To apply this optimisation one often needs to observe carefully on those mathematical constructions and find out the common sub-expressions.

IV. CODE OPTIMISATION IN FINANCIAL PROGRAMMING

Option pricing is at the core of many computational finance problems. Its computational routine should be written in a way that is highly efficient. For instance, to compute implied volatilities option pricing routine is repeatedly called by a root-hunting procedure that compares the calculated option price with the market price. For Monte Carlo simulations, because of the large number of generated scenarios, the time needed by the computation is usually long. For these reasons, optimisations in source code level, besides parallelisation, is often necessary in order to shorten the execution times of the computational finance procedures.

One of the many optimisation techniques we find useful is to use simple data structures. For the binomial tree method, although a tree is a two-dimensional structure, we do not have to explicitly build a tree in memory. Instead, an one-dimensional array is sufficient to store the option values represented by the nodes under processing. All the computations can be done in an one-dimensional array as Fig. 2 shows. Maintaining and traversing an one-dimensional array is much faster than working with a two-dimensional tree.

Another category of code optimisation often applicable to financial code is avoiding repetitive computation on common sub-expressions. In many cases, because of the way those mathematical expressions are constructed there are common parts in them. To save computational time we can compute the common part once and save its value and re-use this

value in subsequent computations.