Parallelizing a Black-Scholes Solver based on Finite Elements and Sparse Grids

Hans-Joachim Bungartz, Alexander Heinecke, Dirk Pflüger, and Stefanie Schraufstetter
Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany
Email: {bungartz,heinecke,pflueged,schraufs}@in.tum.de

***Abstract*—We present the parallelization of a sparse grid finite element discretization of the Black-Scholes equation, which is commonly used for option pricing. Sparse grids allow to handle higher dimensional options than classical approaches on full grids, and can be extended to a fully adaptive discretization method. We introduce the algorithmical structure of efficient algorithms operating on sparse grids, and demonstrate how they can be used to derive an efficient parallelization with OpenMP of the Black-Scholes solver. We show results on different commodity hardware systems based on multi-core architectures with up to 8 cores, and discuss the parallel performance using Intel and AMD CPUs.**

***Keywords*-Black-Scholes; option pricing; sparse grids; finite elements; multi-core; OpenMP**

I. Introduction

Pricing options has been of ongoing interest in the last few decades. Financial products still become more and more complex, varying in the type or number of underlyings, or in the exercise modalities—just consider the European, American or Asian option, and variants like the Bermudan option. Since its introduction in 1973, the Black-Scholes equation has been of central interest pricing European as well as American call and put options.

As in general no closed form solutions are available, options have to be priced either stochastically or numerically. In the former case, Monte-Carlo methods are frequently employed, as they are easy to use and implement—independent of the number of dimensions. On the other hand, they exhibit slow convergence rates, scaling only $\mathcal{O}(1/\sqrt{N})$, with N being the number of random samples, and Greeks are costly to compute.

In the latter case, the partial differential equation (PDE) can be discretized via finite differences (FD) or finite elements (FE), and the resulting linear system of equations is then solved. This way, Greeks can be derived easily, and higher convergence rates are achieved—at least in low-dimensional settings. For higher-dimensional options, discretization methods based on regular grids suffer the so-called curse of dimensionality, as the number of grid points that are needed for error reduction depends exponentially on the dimensionality.

In our approach, we discretize the Black-Scholes equation with finite elements on sparse grids, which enable us to reduce the number of grid points that are needed significantly, breaking the curse of dimensionality to some extent. Furthermore, sparse grids can be extended to allow for adaptivity, spending only grid points where it is really necessary. As this is traded for sophisticated algorithmic structures, the direct sparse grid method has not been applied to the Black-Scholes equation yet.

Compared to methods on classical full grids, more computations per degree of freedom are needed. This requires algorithms to exploit the potential of modern commodity hardware which is heading towards multi- and many-core systems. We therefore parallelized a Black-Scholes solver, exploiting the recursive structure of algorithms working on sparse grids. To this end, we make use of OpenMP 3.0's task concept, and evaluate the parallel performance on both Intel and AMD multi-core architectures.

In this paper, we present an efficient parallelization for solving the Black-Scholes equation with finite elements on sparse grids. To this end, we state the Black-Scholes equation in the next section, introduce the sparse grid FE discretization, and describe the algorithmical structures necessary to solve it. In Sect. III, we show how to efficiently parallelize algorithms working on sparse grids on shared memory systems. Section IV briefly presents some numerical results before focusing on the parallel performance on several hardware architectures with different characteristics. Section V finally summarizes and concludes our work.

II. The Black-Scholes Equation on Sparse Grids

In this paper, we focus on option pricing based on the Black-Scholes equation. The Black-Scholes model is one of the most important concepts in mathematical finance and assumes that the underlying assets follow a geometric Brownian motion. It forms the basis for the numerical pricing of many options, e.g. when combining it with Theta calculus [1], [2] which is a mathematical representation for the description of sequential processes and financial contracts. In the following, we formulate the Black-Scholes equation, discretize it with finite elements, introduce sparse grids in this context and describe the algorithmical structures on which the parallelization is based on.

978-1-4244-6534-7/10/$26.00

A. Black-Scholes with Finite Elements

The multi-dimensional Black-Scholes equation with which the price of a European basket option on d assets at a time T can be evaluated is given by

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sum_{i,j=1}^{d} \sigma_i \sigma_j \rho_{ij} S_i S_j \frac{\partial^2 V}{\partial S_i \partial S_j} + \sum_{i=1}^{d} \mu_i S_i \frac{\partial V}{\partial S_i} - rV = 0\,. \quad (1)$$

Here, $V(t, \mathbf{S})$ denotes the value of the option, $t \in [0, T]$ the forward time, $\mathbf{S} = (S_1, \ldots, S_d)$ the stock values, σ_i the volatilities, ρ_{ij} the asset correlations, μ_i the drifts and r the risk-free interest rate. The end condition is defined by the payoff function that depends on the strike K, for example by

$$V(T, \mathbf{S}) := \begin{cases} \max\{K - \frac{1}{d}\sum_{i=1}^{d} S_i,\, 0\}, & \text{put option} \\ \max\{\frac{1}{d}\sum_{i=1}^{d} S_i - K,\, 0\}, & \text{call option.} \end{cases} \quad (2)$$

Since the PDE has to be solved backwards due to its end condition (2), we additionally define the backward time $\tau := T - t$ and consider $V(\tau)$ instead of $V(t)$ in the following.

There are different numerical methods for solving Eq. (1). The finite difference method is simple to implement but, usually, it has stronger regularity assumptions than the finite element method. It is also less flexible concerning the discretization. By contrast, the finite element method can be easily applied to adaptive meshes and used with different basis functions. Thus, when choosing a hierarchical basis, the finite element method fits well in the concept of sparse grids which is described in Sect. II-B.

Applying the finite element method to (1), transformed in backward time τ, we obtain the weak form

$$\frac{\partial}{\partial \tau}\langle V, w\rangle_{L^2(\Omega)} + \frac{1}{2}\sum_{i,j=1}^{d} \sigma_i \sigma_j \rho_{ij} \langle S_i S_j \frac{\partial V}{\partial S_i}, \frac{\partial w}{\partial S_j}\rangle_{L^2(\Omega)} - \sum_{i=1}^{d} \left(\mu_i - \frac{1}{2}\sum_{j=1}^{d} \sigma_i \sigma_j \rho_{ij}(1 + \delta_{ij}) \right) \langle S_i \frac{\partial V}{\partial S_i}, w\rangle_{L^2(\Omega)} - r\langle V, w\rangle_{L^2(\Omega)} = 0\,, \quad (3)$$

with Kronecker's delta δ_{ij}, the standard L^2 scalar product $\langle \cdot, \cdot \rangle_{L^2(\Omega)}$ on a domain Ω, and $w(\mathbf{S})$ denoting a test function. Here, the boundary terms have already been neglected since we assume Dirichlet boundaries which are given by the discounted payoff function, so that the test functions $w \in H_0^1(\Omega)$ vanish at the boundary $\partial\Omega$.

For the discretization in space, we use the well-known Galerkin projection combined with a space of piecewise linear hierarchical basis functions which is introduced in the next section. The subsequent discretization in time can be done for example with the implicit Euler method or with the scheme of Crank and Nicholson.

B. Discretization with Sparse Grids

To obtain a spatial finite element discretization, we follow the Ritz-Galerkin approach, choose a suitable basis $\{\varphi_i(\mathbf{S})\}_{i=1}^{N}$ of N basis functions associated to grid points, and restrict the problem to a finite dimensional space V_N spanned by the basis functions. We are thus interested in solutions

$$u(\mathbf{S}, \tau) = \sum_{q=1}^{N} \alpha_q(\tau)\varphi_q(\mathbf{S}) \quad (4)$$

for a certain time τ. In the following, we consider the space of piecewise d-linear functions with a mesh width of $h_n := 2^{-n}$ in every dimension for some discretization level $n \in \mathbb{N}_0$.

Unfortunately, the so-called curse of dimensionality severely restricts the number of dimensions that can be tackled: Using a regular grid with $h_n^{-1} = 2^n$ grid points in one dimension leads to $h_n^{-d} = (2^n)^d$ grid points in d dimensions. The exponential dependency on the dimensionality d makes it infeasible to treat more than four dimensions for reasonable values of n.

Here sparse grids come into play. Sparse grids, as introduced in 1990 for the solution of PDEs [3], break the curse of dimensionality to some extent. For sufficiently smooth functions, they enable to reduce the number of grid points significantly to $\mathcal{O}(h_n^{-1}\log(h_n^{-1})^{d-1})$, while keeping almost the same convergence rates as in the full grid case. Sparse grids have meanwhile been employed in various settings, see [4] and the references cited therein. More recent work on sparse grids includes stochastic and non-stochastic partial differential equations in various settings [5], [6], [7], as well as applications in economics [8], [9], regression [10], [11], classification [12], [13], [14], fuzzy modeling [15], and more.

Sparse grids are based on a tensor product construction of hierarchical basis functions. We restrict ourselves in the following to the space of piecewise d-linear functions, and formally define it on the d-dimensional unit-hypercube, i.e., $\Omega = [0, 1]^d$. For other rectangular domains, we have to scale them accordingly. The one-dimensional basis functions are based on the standard hat function $\varphi(x)$,

$$\varphi(x) := \max(1 - |x|, 0)\,, \quad (5)$$

from which we then derive one-dimensional hat basis functions by dilatation and translation,

$$\varphi_{l,i}(x) := \varphi(2^l x - i)\,, \quad (6)$$

which depend on a level $l \in \mathbb{N}$ and an index $i \in \mathbb{N}$, $0 \le i \le 2^l$. The basis functions have local support and are centered at grid points $x_{l,i} = 2^{-l}i$, basis functions centered on $\partial\Omega$

have to be restricted to Ω. They are then extended to the d-dimensional case via a tensor product approach,

$$\varphi_{\mathbf{l},\mathbf{i}}(\mathbf{x}) := \prod_{j=1}^{d} \varphi_{l_j,i_j}(x_j)\,, \tag{7}$$

where $\mathbf{l}$ and $\mathbf{i}$ denote multi-indices indicating level and index for each dimension, $\mathbf{0} := (0,\ldots,0)$, $|\mathbf{l}|_1 = \sum_{j=1}^{d} l_j$, and relations like $\mathbf{0} < \mathbf{l}$ are to be treated component-wise. We can then define the hierarchical increments $W_\mathbf{l}$, spanned by the basis

$$\Phi_{W_\mathbf{l}} := \left\{\varphi_{\mathbf{l},\mathbf{i}}(\mathbf{x}) :\ i_j = 1,\ldots,2^{l_j}-1,\ i_j \text{ odd}, 1 \le j \le d\right\} \tag{8}$$

of which the basis functions spanning a $W_\mathbf{l}$ have supports of the same size and shape with pairwise disjoint interiors.

Considering sufficiently smooth functions, we can now select those subspaces $W_\mathbf{l}$ a priori which contribute most to the overall solution. A continuous optimization problem with respect to both the L^2- and the L^∞-norm (see [4] for details and derivations) finally results in the sparse grid space

$$V_n^{(1)} := \bigoplus_{|\mathbf{l}|_1 \le n+d-1} W_\mathbf{l}\,, \tag{9}$$

a direct sum of subspaces $W_\mathbf{l}$, and in functions

$$u(\mathbf{S},\tau) = \sum_{\mathbf{l},\mathbf{i}} \alpha_{\mathbf{l},\mathbf{i}}(\tau)\varphi_{\mathbf{l},\mathbf{i}}(\mathbf{S}) \quad \in V_n^{(1)} \tag{10}$$

with hierarchical coefficients $\alpha_{\mathbf{l},\mathbf{i}}$, $|\mathbf{l}|_1 \le n+d-1$, which are denoted as surpluses; see Fig. 1 in Sect. II-C for a sparse grid of level 4 in two dimensions. To be able to spend grid points on the boundary we extend the one-dimensional basis on level 1 by two basis functions on level 0 which are located on the boundary, $\varphi_{0,0}$ and $\varphi_{0,1}$.

Substituting V by u in (3) and choosing $V_n^{(1)}$ as both ansatz and test space of the finite element method, we get the weak formulation (3) discretized with the hierarchical basis functions $\varphi_{\mathbf{l},\mathbf{i}}$. It can be written in matrix notation as

$$\begin{aligned} B\frac{\partial}{\partial\tau}\alpha(\tau) = &-\frac{1}{2}\sum_{i,j=1}^{d}\sigma_i\sigma_j\rho_{ij}C\alpha \\ &+\sum_{i=1}^{d}\left(\mu_i - \frac{1}{2}\sum_{j=1}^{d}\sigma_i\sigma_j\rho_{ij}(1+\delta_{ij})\right)D\alpha \\ &+ rB\alpha \end{aligned} \tag{11}$$

with $B_{p,q} = \langle\varphi_p,\varphi_q\rangle_{L^2(\Omega)}$, $C_{p,q} = \langle S_j\frac{\partial\varphi_p}{\partial S_j}, S_i\frac{\partial\varphi_q}{\partial S_i}\rangle_{L^2(\Omega)}$, and $D_{p,q} = \langle\varphi_p, S_i\frac{\partial\varphi_q}{\partial S_i}\rangle_{L^2(\Omega)}$.

Equation (11) can now be solved applying a suitable discretization in τ, using the payoff function (2) to obtain initial values at time step $\tau = 0$.

Up to now, only the so-called combination technique has been used to solve the Black-Scholes equation using sparse grids [8], rather than the direct FE scheme introduced above. The sparse grid combination technique (which is sometimes called sparse grid collocation method) is based on the fact that, when interpolating a function, the sparse grid solution can be obtained via a linear combination of solutions on coarser full grids. In non-interpolation tasks, e.g. when solving PDEs, the combination technique and the direct finite-element technique may exhibit different characteristics. Examples have been found where the combination technique does not converge, and the so-called optimized combination technique has been developed to circumvent this [16].

Nevertheless, the combination technique has usually been used, as it is much simpler to implement and to understand since full grids are common in many applications. Additionally, the $\mathcal{O}(d\,(\log h_n^{-1})^{d-1})$ partial solutions of size $\mathcal{O}(h_n^{-1})$ can be obtained completely in parallel, and efficient out-of-the-box tools like multigrid methods can be used to solve them. On the other hand, extending to higher-dimensional settings requires to use adaptive methods. The combination technique only allows dimensionally adaptive approaches, whereas the direct sparse grid technique (solving a single problem of size $\mathcal{O}(h_n^{-1}(\log h_n^{-1})^{d-1})$) has the advantage that it can be used in a straightforward way to explore and exploit the domain fully adaptively. The adaptive sparse grid technique therefore enables to spend more grid points close to the singularity of the payoff function, e.g. The drawback is that more sophisticated algorithms are necessary to keep efficiency, see Sec. II-C.

Especially the matrices in (11) should not be assembled directly. Using full grids and the common nodal basis, the FE discretization leads to sparsely populated matrices, as only few basis functions overlap. In a hierarchical basis, basis functions overlap with all their ancestors and descendants in the hierarchical structure, leading to a significant fill-in of the corresponding, hence densely populated matrices. But, due to the tensor product structure, algorithms that apply one of the matrices to a vector can be formulated which scale only linearly in the number of grid points. This allows the usage of iterative methods, such as BiCGSTAB.

In the following section, we roughly sketch the algorithmic structures of such matrix applications, as they are necessary for the parallelization schemes we will address later.

C. Algorithmic Structures

Applying the matrices in (11) to a vector can be done efficiently by multiple traversals of the hierarchical tree structure of the sparse grid. For the sake of simplicity, we focus on the Matrix B, which is simpler than the other ones, but complex enough so that all important algorithmic ideas can be shown for it. The main idea is that, due to the tensor product structure of the underlying basis, the application of a matrix derived in d dimensions can be decomposed

into multiple applications of corresponding one-dimensional matrices.

The scheme of applying a one-dimensional operator to all one-dimensional sub-grids of a d-dimensional grid in a certain dimension is called the unidirectional principle, and has first been described in [17]. It is usually repeated for all dimensions, one after the other, working on updated values in each step. The probably simplest algorithms in this respect are the hierarchization and dehierarchization, i.e., the transformation of a vector containing function values at all grid points to the vector of hierarchical surpluses and vice versa. For operators that are more complicated, as it is the case for our matrices, the one-dimensional applications have to be split into so-called Up- and Down-parts, resulting into the algorithmically slightly more complex *UpDown* scheme.

We will first consider the one-dimensional case, neglecting the basis functions on level 0, which have to be handled separately as a special case. Let the basis functions φ_j be ordered in any way so that $j < k$ if basis function φ_j has a smaller level (is hierarchically higher) than φ_k; a breadth-first traversal of the binary tree of basis functions suffices, for example. The matrix-vector product can then be computed in one sweep, recursive in the level l: On the way from the root ($l = 1$) down the tree (*Down*), the contributions of all hierarchically higher basis functions can be gathered for each node, on the way back up (*Up*) the contribution of all lower ones. If we treat both parts separately, we split the matrix B into its lower (B^{Down}) and upper (B^{Up}) triangular sub-matrices and its diagonal matrix (B^{Diag}) which is usually treated together with one of the former ones (i.e. either during the Up or the Down traversal, usually during Down),

$$B\alpha = B^{\text{Down}}\alpha + B^{\text{Diag}}\alpha + B^{\text{Up}}\alpha\,. \tag{12}$$

In other words, arriving at the basis function φ_j during Down, we compute $\langle \varphi_j, \sum_{k \le j} \varphi_k \rangle_{L^2(\Omega)}$, during Up $\langle \varphi_j, \sum_{k > j} \varphi_k \rangle_{L^2(\Omega)}$, and together $(B\alpha)_j$.

There are several reasons to regard the Up and Down parts separately. First, Down can be derived easily as the important information about all hierarchically higher basis functions for a given basis function to compute the Down-part of the scalar product can be gathered descending from the root. Up is generally less intuitive, but can be derived out of Down in a systematic way, even for non-symmetric matrices as the matrix D, but which goes beyond the scope of this paper.

More important in this context is, as it determines the algorithmic structure, that in a d-dimensional setting the Up and Down parts have to be split and treated independently: Iterating the one-dimensional applications over all dimensions requires to work on suitably updated values to be able to compute the tensor product interrelation between the different dimensions. Due to the sparse structure of the sparse grid, all Ups have to be performed before any Down operation when alternating the dimension: the transfer of intermediate contributions between two grid points can only occur via common ancestors in the hierarchical structure. Figure 1 depicts the application of both Up and Down in one dimension for a two-dimensional sparse grid as well as the algorithmic structure of the UpDown algorithm. The latter starts with dimension d, recursing until dimension 1 is reached. In the last dimension, only Up and Down are computed, stopping the recursion.

The parallelization, which is discussed in the next section, is based on the recursive structure of the UpDown algorithm. Note that even though the matrices C and D represent operators that are more complicated, they share the same algorithmic structure and can be handled in the same way, applying one-dimensional versions of Up and Down for each dimension.

III. Parallelization

Dual- and Quad-Core processors have already become the standard CPUs in nearly all kinds of modern computers ranging from notebooks to high performance machines. In the near future, multi-core and many-core (i.e. with more that 16 cores) processors will be the dominating architecture. Furthermore, Intel will release its first high performance graphics accelerator in the beginning of 2010, codenamed "Larrabee", which is built out of standard x86 cores and can be regarded as the first many-core CPU, as a few dozen cores can be expected. Thus, algorithms and implementations that scale well on such shared memory platforms are urgently needed. In this section we demonstrate how the UpDown scheme introduced above can be parallelized on such architectures with a platform-independent approach.

To reach this end, we decided to use OpenMP as the tool for parallelization: Practically all available compilers, free and commercial ones, provide OpenMP support on various platforms. As demonstrated in [18], the OpenMP 3.0's task concept, introduced in 2008, allows a straightforward parallelization of nested recursions which are the basic building blocks of algorithms working on sparse grids.

The new compiler directives introduced in the OpenMP 3.0 standard permit a simple and flexible parallelization of recursive algorithms: Only the parts of the application that should be executed in parallel have to be specified. Additionally, the number of threads, which the application should use, can be specified. Otherwise, the OpenMP runtime system automatically detects the number of available cores and starts threads in the same quantity.

This provides a great enhancement in parallel programming: all synchronization and load balancing tasks are done automatically by the OpenMP runtime environment. As already shown in [18], this does not result in a huge overhead if the parallel jobs provide "enough" computational complexity. The results in Sect. IV show further that very high parallel efficiencies can be reached solving the Black-Scholes equation on sparse grids.

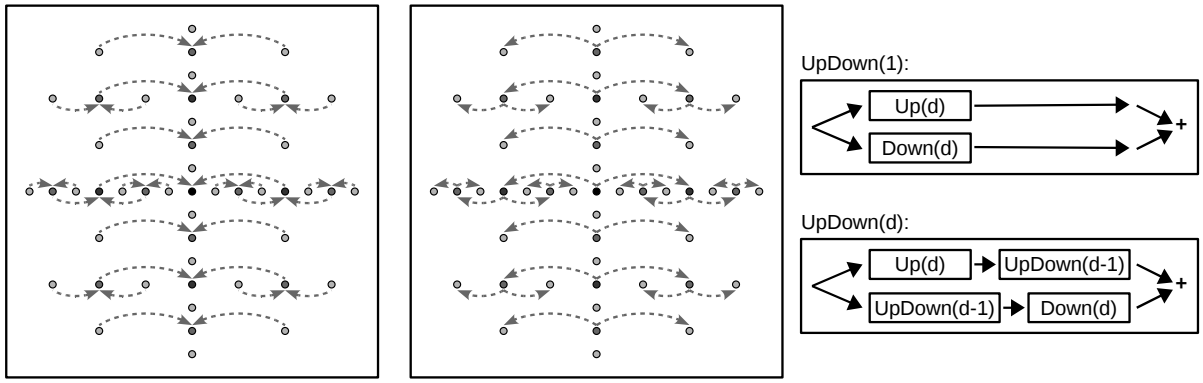

Figure 1. Up (left) and Down (middle) in the first dimension for a two-dimensional sparse grid, showing the one-dimensional tree traversals; the algorithmic structure of the UpDown scheme (right), recursing in the dimension d

The following code example demonstrates how the implementation of the UpDown algorithm can be parallelized using the OpenMP 3.0 task concept. This scheme works for the application of all matrices in (11). For the matrix B, computing plain L^2 scalar products, it is sufficient to provide suitable implementations for the one-dimensional `up`- and `down`-procedures. For the other matrices in the multi-dimensional Black-Scholes equation, a slightly more general UpDown scheme has to be used, allowing to execute different `ups` and `downs`, depending on the current dimension. In the routine's first part, the recursive calls for Up and Down are determined, see also the UpDown scheme in Fig. 1. Both can be executed in parallel, due to the fact that the total result of the operation is given by the sum of all one-dimensional Up and Down calculations. The second branch of the `if` statement considers the last dimension, ending the recursion. As no implicit barriers are given at the end of one task block, an explicit barrier has to be declared by the `#pragma omp taskwait` directive, which synchronizes all tasks that have been started since the last barrier.

```
Vector updown(Vector α, int d) {
  // create result vector b and auxiliary vector c
  Vector b(N), c(N)
  // recursion in remaining dimensions d, ..., 1 :
  if (d > 1) {
    # pragma omp task shared (α, b) {
      b = updown( up(α, d), d − 1 )
    }
    # pragma omp task shared (α, c) {
      c = down( updown(α, d − 1), d )
    }
    # pragma omp taskwait
    b = b + c
  }
  // d = 1, end of recursion:
  else {
    # pragma omp task shared (α, b) {
      b = up(α, d)
    }
    # pragma omp task shared (α, c) {
      c = down(α, d)
    }
    # pragma omp taskwait
    b = b + c
  }
  return b
}
```

Listing 1. OpenMP-Parallelization of b=updown(α, d).

Looking at the parallelization scheme, it can be clearly seen that 2^d parallel tasks are created, so there are plenty of jobs available when pricing multi-dimensional options, for example in five dimensions. The parallel performance provided by contemporary multi-processor or multi-core (and many-core in the near future) systems can thus be easily exploited.

IV. Numerical and Performance Results

In this section, we sum up numerical and performance results of our parallel multi-dimensional Black-Scholes solver. We study a test case in different parallel environments, and provide both numerical and performance results. As the test case, we consider a European basket call option with two to five underlying assets. The payoff function we use is given by (2) with strike $K = 1$. For the parameters of the underlying stochastic process, we choose $\sigma_i = 0.4$, $\rho_{i,j} = \delta_{i,j}$, and $\mu_i = r \in \{0,\, 0.05\}$. Since we are interested in the option value for $\mathbf{S} = (1, \ldots, 1)$, we choose the domain $[0, 2]^d$. It turned out that, in this case, the boundaries do not have much effect on the option value any more. To discretize the system of ordinary PDEs in time, we use the method of Crank and Nicholson. All options are priced at physical time $T = 1$ (time to maturity is 1 year) within 10 time steps. In every time step, the stopping condition for the iterative solution of the corresponding linear system is based on the BiCGSTAB's convergence, reducing the relative residual to less than 10^{-5} considering the Euclidean norm.

A. Testing Environments

We tested the parallel Black-Scholes solver on various platforms, showing significantly differing hardware architectures and performance characteristics:

- A mobile Intel Penryn Core2Duo processor (SP9300) running at 2.26 GHz with 4 GB DDR3 main memory.
- A two socket Intel Nehalem system with Intel Xeon X5570 running at 2.93 GHz with 12 GB DDR3 main memory. The CPUs communicate via Intel's *Quick Path Interconnect*.
- A two socket AMD Shanghai system with AMD Opteron 2378 running at 2.4 GHz with 32 GB DDR2 main memory. The CPUs communicate via the *Hypertransport* protocol.

In all multi-socket cases, we restricted ourselves to NUMA systems as both leading companies, Intel and AMD, will only develop and improve these kinds of machines in the future. In a NUMA system every socket is equipped with its own private memory which can be accessed very fast. Fetching memory located in banks attached to other CPUs is a bit slower.

B. Numerical Results

In the following, we present numerical results solving the Black-Scholes equation. Tab. I shows the results for regular sparse grids of several levels in up to 5 dimensions. Additionally, we provide as a comparison the option price obtained by a classic Monte-Carlo (MC) simulation with 10^6 paths for each test case, allowing to verify the option prices obtained with sparse grids.

In up to 4 dimensions, we can observe that the option price converges nicely with increasing grid level. The rather bad convergence in the 5-dimensional case can be ascribed to the low resolution of the grid in the inner part of Ω. On level 6, still only 5,503 grid points are spent in the interesting inner part, whereas 97,282 ones are spent on $\partial\Omega$. Employing adaptivity, which we have to leave for future work, promises to provide much better results. The grid next to the boundary, where the option price is almost linear, can be adaptively coarsened, and the more important inner part of the grid, where the price is at-the-money, can be refined instead.

C. Performance Results

In this section, we finally sum up the performance results we measured on the different hardware architectures. To this end, we compare the parallel execution time on several CPUs with the sequential time measured on a single CPU. We therefore determine the parallel efficiency E_n of the Black-Scholes solver using n threads by

$$E_n := \frac{t_1}{t_n \cdot n}, \tag{13}$$

with t_1 denoting the execution time on a single CPU and t_n the execution time using n CPUs with n threads. In the following, we only focus on the maximal level shown in Sect. IV-B for which we obtained the best numerical results.

	2 underlyings		3 underlyings	
level	$r = 0.00$	$r = 0.05$	$r = 0.00$	$r = 0.05$
2	0.12877	0.15130	0.11507	0.12815
3	0.12063	0.14080	0.10857	0.12836
4	0.12126	0.14035	0.09898	0.11659
5	0.11580	0.13498	0.08941	0.10809
6	0.11483	0.13398	0.09305	0.11128
7	0.11303	0.13223	0.09335	0.11160
8	0.11217	0.13137	0.09276	0.11104
9	0.11105	0.13026		
10	0.11065	0.12987		
MC	0.11413	0.13698	0.09382	0.11822

	4 underlyings		5 underlyings
level	$r = 0.00$	$r = 0.05$	$r = 0.00$
2	0.10355	0.09428	0.09000
3	0.14326	0.16028	0.26968
4	0.10003	0.11462	0.20286
5	0.07122	0.08849	0.07565
6	0.07313	0.09145	0.05962
7	0.08111	0.09751	
MC	0.08189	0.10654	0.07348

Table I
COMPARISON OF OPTION PRICES FOR SPARSE GRIDS OF SEVERAL LEVELS AND FOR MC SIMULATIONS WITH 10^6 PATHS, $r \in \{0, 0.5\}$ FOR DIFFERENT NUMBERS OF UNDERLYINGS.

First, we analyze the performance of the mobile CPU, the Intel SP9300. Tab. II shows the serial and parallel execution times and parallel efficiencies. Having only two cores, only two threads have been used. For all options but the 4-dimensional option with $r = 0$, not the whole parallel performance which is theoretically possible can be exploited. Considering the example code above, it can be seen that the summation of the two result vectors is not computed in parallel; as both cores reach the synchronization barrier at the same time, one of them is idle until the summation has been completed. For the 4-dimensional option with

option type		1 thread	2 threads	
dim. d	r	t_1	t_2	E_2
2	0.00	730 s	390 s	0.94
	0.05	747 s	430 s	0.87
3	0.00	3870 s	2010 s	0.96
	0.05	3870 s	2150 s	0.90
4	0.00	32000 s	15440 s	**1.04**
	0.05	32200 s	17270 s	0.93

Table II
PERFORMANCE RESULTS ON THE INTEL CORE2DUO SP9300 (PENRYN) FOR DIFFERENT NUMBERS OF UNDERLYINGS AND THREADS; WALL CLOCK TIMES t_n AND PARALLEL EFFICIENCIES E_n; SUPER-LINEAR EFFICIENCIES MARKED IN BOLD.

$r = 0$, we can observe an exceptionally excellent, super-linear speedup. It is most likely caused by the processor's level 2 cache which is shared by both cores. One core might have fetched memory that the other core needs as well. This saves cache misses and therefore expensive memory access with higher latency, resulting in faster execution times. Parallelizing the summation as well did not improve the execution times, as this operation is extremely memory bounded.

The performances of the two socket server systems are examined in Tab. III and IV. (Due to limited computing time on the test platforms, we just considered 2 and 8 threads and $r = 0$ for the $5d$ option.) Again, a similar picture can be

option type		1 thread	2 threads	
dim. d	r	t_1	t_2	E_2
2	0.00	580 s	300 s	0.97
	0.05	610 s	310 s	0.98
3	0.00	3060 s	1540 s	0.99
	0.05	3060 s	1540 s	0.99
4	0.00	26860 s	12100 s	**1.11**
	0.05	26900 s	12150 s	**1.11**
5	0.00	176700 s		

option type		4 threads		8 threads	
dim. d	r	t_4	E_4	t_8	E_8
2	0.00	220 s	0.66	220 s	0.33
	0.05	230 s	0.66	230 s	0.33
3	0.00	950 s	0.81	810 s	0.47
	0.05	970 s	0.79	810 s	0.47
4	0.00	6960 s	0.97	4760 s	0.71
	0.05	7000 s	0.96	4790 s	0.70
5	0.00			23600 s	0.94

Table III
PERFORMANCE RESULTS ON THE INTEL XEON X5570 (NEHALEM) FOR DIFFERENT NUMBERS OF UNDERLYINGS AND THREADS; WALL CLOCK TIMES t_n AND PARALLEL EFFICIENCIES E_n; SUPER-LINEAR EFFICIENCIES MARKED IN BOLD.

seen. We therefore focus only on a few observations that have to be remarked. It can be clearly seen that there are only four tasks for two-dimensional options, and we thus obtain the same execution times using 4 or 8 cores. For the four-dimensional option we obtain super-linear efficiency again, this time for both $r = 0$ and $r = 0.05$. However, the Shanghai's overall performance in terms of execution time, is extremely strange, taking more than twice as long as the Nehalem (with just about 20% slower clock frequency). Both Intel and AMD use prefetching, branch prediction and out-of-order mechanisms to improve the performance of the processor. Of course, both differ in their hardware implementation and the underlying algorithms, but this cannot be the only reason for such big differences. One possible explanation is the processors' cache-associativity. Intel uses a 32 KB 8-way level one cache [19], AMD a 64 KB 2-way level one cache [20]. This implies that Intel's cache is

option type		1 thread	2 threads	
dim. d	r	t_1	t_2	E_2
2	0.00	1230 s	630 s	0.98
	0.05	1270 s	660 s	0.96
3	0.00	6010 s	3020 s	0.99
	0.05	6010 s	3010 s	0.99
4	0.00	46250 s	23150 s	0.99
	0.05	51030 s	25500 s	1.00
5	0.00	272500 s		

option type		4 threads		8 threads	
dim. d	r	t_4	E_4	t_8	E_8
2	0.00	470 s	0.65	470 s	0.33
	0.05	490 s	0.65	490 s	0.32
3	0.00	2070 s	0.73	1580 s	0.48
	0.05	2060 s	0.73	1580 s	0.48
4	0.00	14480 s	0.79	11200 s	0.52
	0.05	16080 s	0.80	11000 s	0.58
5	0.00			52400 s	0.65

Table IV
PERFORMANCE RESULTS ON THE AMD OPTERON 2378 (SHANGHAI) FOR DIFFERENT NUMBERS OF UNDERLYINGS AND THREADS; WALL CLOCK TIMES t_n AND PARALLEL EFFICIENCIES E_n.

four times as efficient in case of equally distributed memory accesses, and that less cache lines replacements are needed. In our sparse grid implementation we use hash maps to store and address coefficients for grid points, so we can assume that exactly this is the case. Another difference worth noting concerns the parallel efficiency. The Shanghai scales significantly worse than the Nehalem, especially in higher dimensional settings where 4 or more threads can be used. This is quite likely due to the fact that Intel's QPI (Quick Path Interconnect) allows all eight cores to access the memory faster than AMD's Hypertransport.

V. CONCLUSION AND OUTLOOK

In this work, we introduced the direct finite element discretization of the Black-Scholes equation using sparse grids. We described the main algorithmic building block, the UpDown scheme, which allows the efficient application of the matrices resulting from the FE discretization. Taking advantage of the task concept of the OpenMP 3.0 standard, we showed an efficient parallelization of the sparse grid Black-Scholes solver, which is well suited for modern multi-core commodity systems. We presented numerical results for basket options with up to five underlyings. Studying the parallel performance on three multi-core architectures, we obtained good parallel efficiencies on Intel architectures with even super-linear performance in four-dimensional settings.

Even though the parallelization methods presented in this paper have been developed to solve the Black-Scholes equation, they are not limited to this setting but can also be applied to different parabolic or elliptic partial differential equations (to the log-transformed Black-Scholes PDE, which

has constant coefficients and models the log-price of the option, e.g.)—and even in applications such as regularization or regression in Data Mining.

Regarding the numerical results, employing adaptivity promises to improve the accuracy significantly, especially for more than four underlyings. For regular sparse grids examined so far, too many grid points are spent on the domain's boundary and not in the interesting interior where the price is at-the-money. It can be expected that insight obtained using adaptive sparse grids in other applications can be also transferred to the Black-Scholes FE discretization. Note that adaptivity might have effects on the parallelization, as it results in non-symmetric grids.

Considering the parallelization, further work has to be done. For example, the super-linear parallel efficiency has to be studied further. This way, it might be understood why the super-linear behaviour could not be encountered on AMD systems. We postulated that that AMD's level-one cache, which is only 2-way associative, is responsible for the bad overall performance with respect to computation time, compared to the one achieved on Intel's systems. A closer examination of the CPU's performance counters will be able to clarify this: A much higher number of cache misses in comparison would confirm the postulation.

Furthermore, the sparse grid Black-Scholes solver has to be ported to new systems such as Larrabee, Nehalem EX or four socket six core Opteron systems.

ACKNOWLEDGEMENTS

The multi-core platforms used for measurements in this paper were provided by the chair for parallel computer architectures (LRR) at the TU München. We want to thank our colleagues at the LRR, especially Carsten Trinitis and Thomas Müller, for giving us access to their machines and their valuable support. We would also like to thank Stefan Zimmer for fruitful sparse grid discussions. The fourth author additionally acknowledges support from the German Federal Ministry of Education and Research (BMBF).

REFERENCES

[1] S. Dirnstorfer, "An introduction to theta-calculus," *SSRN eLibrary*, 2005.

[2] S. Schraufstetter and J. Benk, "A general pricing technique based on theta-calculus and sparse grids." ENUMATH, Uppsala, 2009, accepted.

[3] C. Zenger, "Sparse grids," in *Parallel Algorithms for Partial Differential Equations*, ser. Notes on Numerical Fluid Mechanics, W. Hackbusch, Ed., vol. 31. Vieweg, 1991, pp. 241–251.

[4] H.-J. Bungartz and M. Griebel, "Sparse grids," *Acta Numerica*, vol. 13, pp. 147–269, 2004.

[5] T. von Petersdorff and C. Schwab, "Sparse finite element methods for operator equations with stochastic data," *Appl. Math.*, vol. 51, no. 2, pp. 145–180, 2006.

[6] B. Ganapathysubramanian and N. Zabaras, "Sparse grid collocation schemes for stochastic natural convection problems," *J. Comp. Phys.*, vol. 225, no. 1, pp. 652–685, 2007.

[7] G. Widmer, R. Hiptmair, and C. Schwab, "Sparse adaptive finite elements for radiative transfer," *J. Comp. Phys.*, vol. 227, pp. 6071–6105, Jun. 2008.

[8] C. Reisinger and G. Wittum, "Efficient hierarchical approximation of high-dimensional option pricing problems," *SIAM J. Scientific Computing*, vol. 29, no. 1, pp. 440–458, 2007.

[9] M. Holtz, "Sparse Grid Quadrature in High Dimensions with Applications in Finance and Insurance," Dissertation, Institut für Numerische Simulation, Universität Bonn, 2008.

[10] J. Garcke and M. Hegland, "Fitting multidimensional data using gradient penalties and the sparse grid combination technique," *Computing*, vol. 84, no. 1-2, pp. 1–25, April 2009.

[11] J. Garcke, "Regression with the optimised combination technique," in *ICML '06: Proc. 23rd int. conf. Machine learning*. New York, NY, USA: ACM Press, 2006, pp. 321–328.

[12] H.-J. Bungartz, D. Pflüger, and S. Zimmer, "Adaptive sparse grid techniques for data mining," in *Proc. of HPSC'06*. Springer, Jun. 2008, pp. 121–130.

[13] J. Garcke, M. Griebel, and M. Thess, "Data mining with sparse grids," *Computing*, vol. 67, no. 3, pp. 225–253, 2001.

[14] D. Pflüger, B. Peherstorfer, and H.-J. Bungartz, "Spatially adaptive sparse grids for high-dimensional data-driven problems," *J. Complexity*, 2009, accepted.

[15] A. Klimke, R. Nunes, and B. Wohlmuth, "Fuzzy arithmetic based on dimension-adaptive sparse grids: a case study of a large-scale finite element model under uncertain parameters," *Int. J. Uncert. Fuzz. Knowledge-Based Syst.*, vol. 14, pp. 561–577, 2006.

[16] M. Hegland, J. Garcke, and V. Challis, "The combination technique and some generalisations," *Linear Algebra and its Applications*, vol. 420, no. 2–3, pp. 249–275, 2007.

[17] R. Balder, "Adaptive Verfahren für elliptische und parabolische Differentialgleichungen auf dünnen Gittern," Dissertation, TU München, 1994.

[18] A. Heinecke and M. Bader, "Towards many-core Implementation of LU Decomposition using Peano Curves," in *Proc. of UCHPC-MAW'09*. New York: ACM, May 2009, pp. 21–30.

[19] Intel Corporation, "Datasheet: Intel Core2 Duo Processor E8000 and E7000 Series," http://download.intel.com/design/processor/datashts/318732.pdf, Jun. 2009, [Online; accessed 27-Oct-2009].

[20] AMD Inc., "Family 10 AMD Opteron Processor Product Data Sheet," http://support.amd.com/us/Processor_TechDocs/40036.pdf, Jun. 2009, [Online; accessed 27-Oct-2009].