

Optimization of Binomial Option Pricing on Intel MIC Heterogeneous System

Weihao Liang^(✉), Hong An, Feng Li, and Yichao Cheng^(✉)

School of Computer Science and Technology,
University of Science and Technology of China, Hefei 230026, China
{lwh1990, fli168, yichao}@mail.ustc.edu.cn, han@ustc.edu.cn

Abstract. In these years, computerization has been more and more important in the financial area. The computational intensity and real-time constraints of those financial models require high-throughput parallel architectures. In this paper, optimization of widely-used binomial option pricing model has been implemented on the worlds largest super-computer, Tianhe-2. In our work, we employ several optimizing techniques to efficiently utilize the architecture of Intel MIC heterogeneous system to improve the performance. The experimental results show that, compared with the serial implementation, the optimized binomial option pricing achieves 33X speedup on one Intel Xeon CPU and 61X speedup on one Intel Xeon Phi coprocessor. Further experiments on Intel MIC heterogeneous system indicate that our implementation attains a speed-up factor of 254 on one Tianhe-2 computing node.

Keywords: Binomial option pricing · MIC · Parallel process · Heterogeneous system · Optimization

1 Introduction

An option is a contract that gives right to the owner to buy or sell a financial asset or instrument at a specified strike price on or before the expiry date. Binomial option pricing is one of the most popular approaches that values an option using a time-step model [10]. With larger scale of financial problem, the number of option grows rapidly which makes the computation becomes very expensive. This issue often happens when a great number of real-time options need to be revaluated with live data. Hence, it is significant to improve the efficiency of binomial option pricing.

Related work mainly focuses on implementing the binomial option pricing on CPU. Previous researchers have presented parallel solution for binomial option pricing with low performance [17]. Gerbessiotis et al. [8] achieved high speedup with proposed algorithm on Intel Pentium CPU cluster. But it only achieves less than 2% of peak performance of one single node and the performance decreases when scaling to multiple nodes. Zubair et al. [18] and John et al. [16] proposed another algorithm considering the memory hierarchy system to achieve 60% of

the peak performance on 8 UltraSPARCIII processors. But it is not easy to optimize and extend to large CPU cluster. Other works deploy binomial option pricing on GPU. Matthew et al. [6] proposed a GPU-based market value-at-risk estimation algorithm which is suitable for Nvidia GPGPU. But it can only process small-scale Europe options. Although Qiwei et al. [12] and Mehmet et al. [9] also provided other GPU-based solutions for binomial option pricing, it does not get enough efficiency from the hardware.

In this paper, we implement and optimize the binomial option pricing on Intel MIC heterogeneous system which contains Intel Xeon CPU and Intel Phi coprocessor. Our method includes several optimizing techniques such as optimizing compiler options, OpenMP parallelization, vectorization by SIMD and modification of the serial algorithm to efficiently utilize Intel hardware's architecture to further improve the performance. The experimental results demonstrate that, compared with the CPU serial code, the optimized version of binomial option pricing achieves 33X speedup on one Intel Xeon CPU and 61X speedup on one Intel Xeon Phi. Further experiments on heterogeneous system of Tianhe-2 [7] indicate that our implementation attains a speedup of 254 times on one Tianhe-2 computing node.

The rest of the paper is organized as follows. We review the binomial option pricing model in Sect. 2 and introduce the architecture and programming model of Intel MIC heterogeneous system in Sect. 3. In Sect. 4, we present and implement several optimization strategies for binomial option pricing on Intel Xeon CPU and Intel Xeon Phi coprocessor. Detailed experimental results and analysis are provided in Sect. 5 and conclusions of our work is shown in Sect. 6.

2 The Binomial Option Pricing Model

2.1 Binomial Option Pricing Model Theory

The flowchart of the binomial option pricing is illustrated in Fig. 1. The binomial pricing model traces the evolution of the option's key underlying variables in discrete time which is done by a binomial tree [5]. Each node in the tree represents a possible price at a given point of time. The computation is performed iteratively, beginning at each of the leave nodes and then computing backwards through the tree towards the root node. The value of each node which depends on the values of its two child nodes is computed at each stage at that given point of time. In the end, the value of root node is the final result of binomial option pricing.

2.2 Serial Algorithm

Algorithm 1 presents the pseudocode of the serial kernel in binomial option pricing program. There are two main parts: the first one is a loop that initialize each leave node's value in the binomial tree (line 2 – 4); the second one is a nested loop (line 6 – 10). The outer loop is procedure of calculation following the time

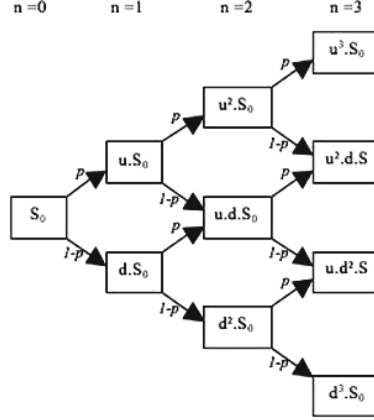


Fig. 1. The flowchart of the binomial option tree.

step from leave node to root node. The inter loop indicates that in each time step, all nodes of that stage will calculate its value by using its two child nodes' values. [13]

Algorithm 1. Serial algorithm of binomial option pricing

```

1: for all options do
2:   for all leave nodes do
3:     Initialize its value
4:   end for
5:
6:   for each time step do
7:     for nodes of each stage do
8:       Calculate its value
9:     end for
10:   end for
11: end for

```

3 Overview of Intel Many Integrated Core (MIC)

3.1 MIC Architecture

Figure 2 shows the microarchitecture of the Intel MIC. The main components of Intel Xeon Phi coprocessor are processing cores, caches, memory controllers, PCIe client logic, and a bidirectional ring interconnect with very high bandwidth. Each core directly connects with a private L2 cache. The memory controllers and the PCIe client logic respectively provide a direct interface to the GDDR memory on the coprocessor and the PCIe bus. In general, all these components are linked together by the ring interconnect [1]. Each core in the Intel Xeon Phi

coprocessor is designed like an x86 processor and providing good programmability [11]. Another important feature of the MIC core is the 512 bit wide vector processing unit (VPU) which supports up to 8 double precision or 16 single precision floating point operations in a single vector instruction.

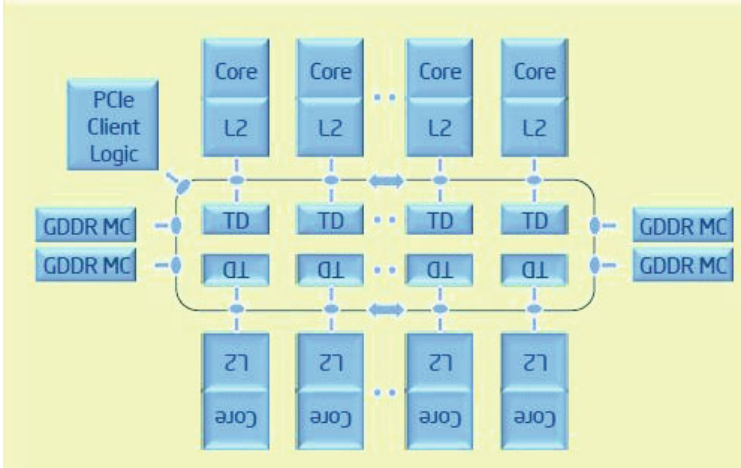


Fig. 2. The MIC microarchitecture.

3.2 MIC Programming Model

Execution of program normally begins on the host CPU and when it reaches some user-defined sections of the code, the corresponding parts are offloaded by the host CPU to the accelerators (MICs). The Intel MIC is based on x86 architecture and therefore standard parallel programming models like OpenMP [3] and MPI [2] can be seamlessly ported to MIC. There are three programming models in MIC: Native mode, symmetric mode and offload mode. Our work only use the offload mode [15]. In the MIC offload mode, the part of code which will be executed on the accelerators is following the offload pragma.

4 Optimizing Strategies

4.1 Optimization on Single CPU

In this section, we propose and implement several optimizing methods to efficiently utilize the features of Intel software and hardware environment.

Optimizing the Compiler Options. Because all the value in the binomial tree is float type or double type, large amount of floating-point data calculation will be performed during the runtime. For Intel CPU's architectures, compilation option *-fp-model* can allow the user to control the optimizations on floating-point data. When the option is turned on, the compiler optimizer removes redundant moves from the FPU registers to memory and back, leaving intermediary results in the FPU stack. With elaborately choosing the level of *-fp-model* option, it will always have good effect on compute-intensive applications' performance with guarantee of the values' safety [4].

Parallelization and SIMD. Because there is no data dependence between all the options. We can simply parallelize the outmost loop in the Algorithm 1 by using OpenMP parallel programming model. After that, we vectorize the inner loop by using AVX-256 SIMD instruction sets to overwrite the corresponding codes which aim at initialize the values of all leave nodes. Because the length of vector process unit (VPU) in Intel Xeon CPU is 256 bit, we can process 4 nodes (each node contains a double-precision floating point number) in one time by using SIMD instructions. In addition, some common factor which will be repeatedly computed in each iteration of the inner loop can be extracted out of the loop to be computed once instead. After the vectorization of the inner loop, we may further unroll the innermost loop. 8 is chosen as the unrolled factor based on some experiments.

Blocking Cache Memory. We can efficiently improve the reuse of data from cache by blocking the accessed data in the innermost loop. The core compute kernel of binomial option pricing is shown in Algorithm 2. The procedure of the core kernel begins from the leave nodes and moves backward in each time step which the values of *Call* array is updated by being computed at a previous step. Eventually, *Call*[0] contains the option price as the final result. Although the compiler is able to automatically vectorize and unroll the *j* loop of the reference code for Intel Xeon CPU, the resulting code still has troubles with unaligned load and loss of SIMD efficiency. Besides, frequently memory access of *Call* array may cause high miss rate in L1 cache which can be monitored from Intel Vtune [14].

Algorithm 2. Compute kernel of binomial option pricing

```

1: for  $n = 0$  to  $N$  do
2:   for  $i = TS$  to  $1$  do
3:     for  $j = 0$  to  $i - 1$  do
4:        $opt[n].Call[j] \leftarrow puByDf * opt[n].Call[j + 1] + puByDf * opt[n].Call[j]$ 
5:     end for
6:   end for
7:    $opt[n].Result \leftarrow opt[n].Call[0]$ 
8: end for

```

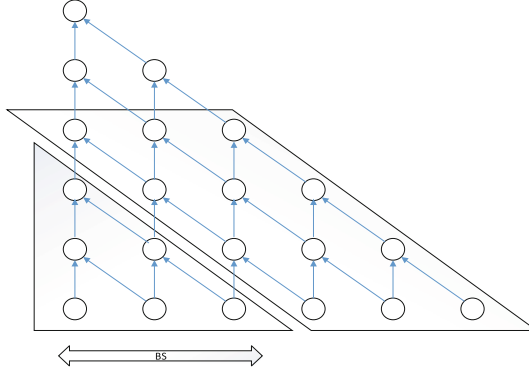


Fig. 3. Cache blocking algorithm.

In order to address the problem above, we present a auto-tuning cache blocking algorithm which shown in Algorithm 3 based on L1 cache size. The algorithm reduces both the working data set and instruction overhead of the computer kernel. First we chose a small or medium input size for this method in order to improve efficiency because we need to run the program more than one time. Then as shown in the outermost loop (line 2), we repeatedly run the compute kernel with increasing block size BS based on the size of a block array which can be allocated in a processors L1 cache. Figure 3 illustrates the key computing kernel (line 5 – 15), we separate the computation into two parts: in the first part (line 6), we read the first blocks values from the *Call* array and reduce it within the memory size of cache (see the lower triangular portion in Fig. 3). Time taken by this part is considered to be negligibly small; in the second part (line 7 – 16), the successive values are read and reduced by BS time steps from the *Call* array, then stored back to $Call[i-BS]$ (the trapezoidal portion in Fig. 3). Some temporary variable like $b1$, $b2$, $b3$ are used to improve reuse of data. So we read each value of *Call* array and store it back only once for every BS time step. In order to take advantage of Intel Xeon CPU's hardware features, we can use extra instructions to prefetch the data that we need during each iteration of inner loop which can further improve the cache's utilization. The entire computation during each BS time step occurs only within the cache which efficiently improves the arithmetic intensity of the code. At the end of each auto-tuning run, the computation time will be recorded and the minimum of all run (line 18 – 21) represents the corresponding best block size which will be chosen as the right parameter to run our compute kernel with normal input size. By applying the cache blocking algorithm with chosen best block size, the miss rate of L1 cache is lower than before which is monitored from Intel Vtune [14]. Overhead brought by this cache blocking algorithm is rather small compared to the benefit of its improvement of data reuse which we can see in further experiments shown in Sect. 5.

Algorithm 3. Auto-tuning cache blocking algorithm

```

1:  $minTime = minBS = MAXINT, btime, etime$ 
2: for  $BS = 16, 32, 64, \dots$  to  $CacheSize/ValueType$  do
3:    $VectorType\ Block[BS], b1, b2, b3$ 
4:    $btime \leftarrow gettime()$ 
5:   for  $n = N, N - BS, \dots$  to  $BS$  do
6:      $\dots // \text{Calculate lower triangular portion}$ 
7:     for  $i = BS$  to  $n$  do
8:        $b1 \leftarrow Call[i]$ 
9:       for  $j = BS - 1$  to  $0$  do
10:         $b2 \leftarrow puByDf * b1 + puByDf * Block[j]$ 
11:         $Block[j] \leftarrow b1$ 
12:         $b1 \leftarrow b2$ 
13:      end for
14:       $Call[i - BS] \leftarrow b1$ 
15:    end for
16:  end for
17:   $etime \leftarrow gettime()$ 
18:  if  $etime - btime < minTime$  then
19:     $minTime \leftarrow etime - btime$ 
20:     $minBS \leftarrow BS$ 
21:  end if
22: end for
23: return  $minBS$ 

```

4.2 Porting to Single MIC

Because Intel Xeon Phi coprocessor (MIC) have more core and longer SIMD width than Intel Xeon CPU, it is obvious that the performance will be significantly improved by porting the computing codes to MIC. So we rewrite the corresponding AVX-256 instruction sets to AVX-512 instruction sets which fits the architecture of MIC. Additionally, MIC provide multiplication-subtraction operation instruction which can increase some degree of the arithmetic intensity of the code. We implement this part with the offload mode of MIC Programming Model.

4.3 Communication Optimization Between CPU and MIC

When the program is applied on the Intel MIC heterogeneous system, communication and cooperative problem between CPU and MIC cannot be neglected. Thanks to no data dependence between the options, the binomial option pricing model is highly scalable. The key communication problem becomes the task partition between CPU and MIC. If the task distribution is unbalanced, the synchronization problem will cause serious bottleneck in performance. To solve this problem, our solution is fine-grained tuning the task partition ratio between CPU and MIC by running the application multiple times with various task partition ratios. In the experiment, we set the ratio of the whole task for CPU from

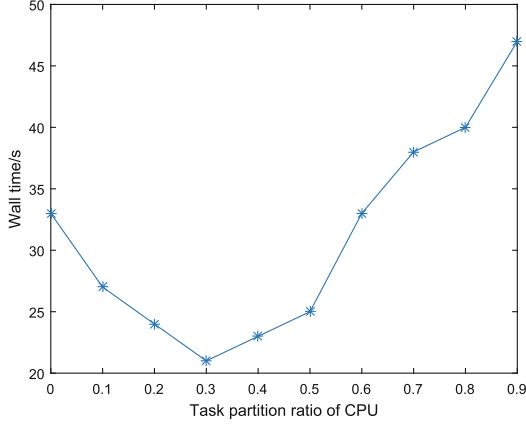


Fig. 4. Performances of different task partition ratio.

0.1 to 0.9 and run the program repeatedly. As we can see in Fig. 4, the optimal ratio is 0.3 when the wall time of whole application is the minimum.

When it comes to large heterogeneous system containing multiple CPUs and multiple MICs, we can deduce the optimal task partition ratio of each CPU and MIC based on the experimental result of Fig. 4. Let e denote the optimal partition ratio for CPU in the system containing one CPU and one MIC and e can be easily got through experiments above. If an Intel MIC heterogeneous system has M CPUs and N MICs, each CPU's best task ratio is r_c and each MIC's best task ratio is r_m . The relationship of r_c and r_m should be represented by:

$$Mr_c + Nr_m = 1 \quad (1)$$

Equation 1 indicates that all CPUs' ratio is the same and so does MICs. The sum of all ratio is 1. Equation 2 is based on the optimal task partition ratio of between one CPU and one MIC. In order to balance the work load between multiple CPUs and MICs, r_c and r_m must meet by:

$$\frac{r_c}{r_m} = \frac{e}{1-e} \quad (2)$$

We can get r_c and r_m after solving the linear equations containing Eqs. 1 and 2.

$$r_c = \frac{e}{Me + N(1-e)} \quad (3)$$

$$r_m = \frac{1-e}{Me + N(1-e)} \quad (4)$$

5 Experimental Results and Analysis

In this section, we first describe the configuration of our hardware platform. And then we present and analysis the performance of binomial option pricing

on one single Intel Xeon CPU with different optimizing strategies. Furthermore, we show the performance on MIC heterogeneous system.

5.1 Hardware Platform Setup

We used one compute node of Tianhe-2 as the experimental hardware platform, which has two Intel Ivy Bridge E5-2692 CPUs and three Intel Xeon Phi 31S1P coprocessors. Each E5-2692 CPU has 12 cores and each 31S1P coprocessor has 57 cores. One compute node of Tianhe-2 can theoretically provide about 5.0 GFLOPs performance.

5.2 Single-CPU Performance

We test the single-CPU performance on the Intel Ivy Bridge CPU with different large problem size. We measure the speedup and Gflops rating of binomial option pricing with different optimizing strategies to the serial algorithm. In the experiment, each optimizing strategy is on the basis of the previous. The performance results are shown in Fig. 5.

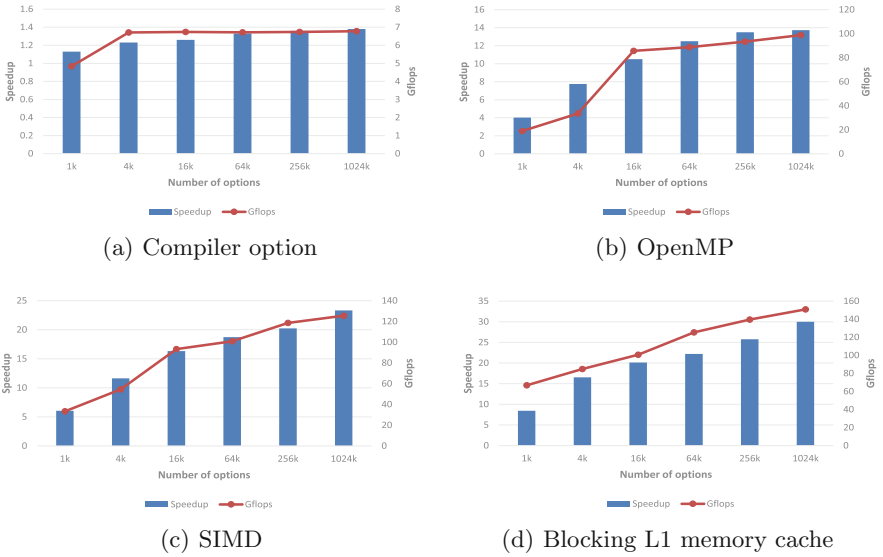


Fig. 5. Speedup and Gflops rating of binomial option pricing with different optimizing strategies.

We can observe from the Fig. 5 that optimizing compiler options improves a little (about 10 %) in performance and parallelization with OpenMP can provide linear speedup (13X) corresponding to the number of cores in CPU. In addition, SIMD offers a very high performance which can achieve extra 10X speedup. As

elaborated in Sect. 4, block size of the cache blocking algorithm has an impact upon the performance. To determine the optimal block size, we test performance of the proposed algorithm for different block sizes which can be seen in Fig. 6. We found the optimal block size to be 128 for $N=4\text{ K}$, 16 K , and 64 K which N is the number of options. By employing the auto-tuning cache blocking algorithm, we can obtain 33X speedup which surpasses previous version by over 10X. From Intel Vtune, we also monitor the L1 cache miss rate decreasing from 0.016 to 0.008, which demonstrates that the cache blocking optimization does really work.

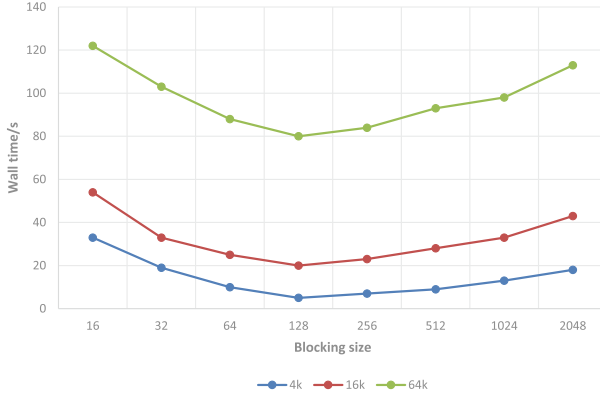


Fig. 6. Performance of cache blocking algorithm for different blocking sizes.

Table 1 lists percentage of the peak performance we are getting for these optimizing methods. N is the number of options. Each optimizing strategy is on the basis of the previous. For large problem sizes, we observe over 50 % of the peak performance, which typically demonstrates that there is a good match of the optimizing algorithm with the underlying architecture of Intel Xeon CPU.

Table 1. Percentage of the peak performance

N	Compiler options	OpenMP	SIMD	Cache block
1 k	1.7 %	6.7 %	11.8 %	23.7 %
4 k	2.3 %	11.9 %	19.3 %	30.1 %
16 k	2.3 %	30.4 %	33.2 %	35.7 %
64 k	2.3 %	31.5 %	35.8 %	44.5 %
256 k	2.4 %	33.3 %	42.1 %	49.6 %
1024 k	2.4 %	35.1 %	44.6 %	54.8 %

5.3 MIC Heterogeneous System Performance

We first conduct the performance on one single Intel Xeon Phi 31S1P coprocessor with MIC offload mode. When it comes to heterogeneous system, task distribution between host (CPU) and device (MIC) becomes important. Our solution is based on the fine-grained tuning the task partition ratio between CPU and MIC which is explained in Sect. 4. One compute node of Tianhe-2 contains two CPUs and three MICs and the optimal task partition ratio of one CPU is 0.11 and one MIC is 0.26 which is conducted by the conclusions of Eqs. 3 and 4 in Sect. 4.

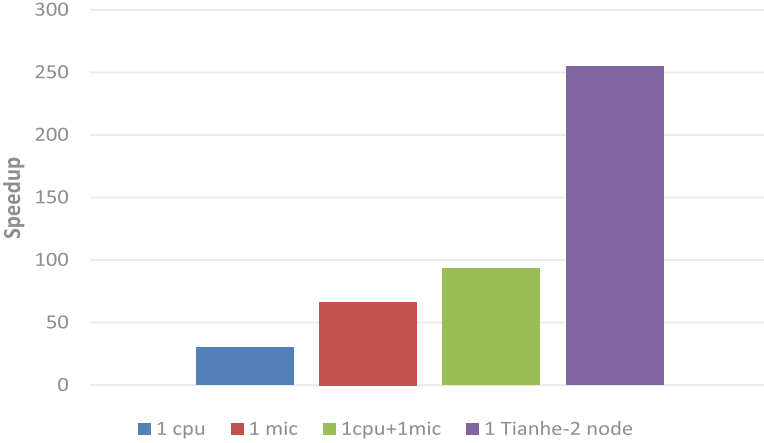


Fig. 7. Performance of Intel MIC heterogeneous system.

The experimental results is illustrated in Fig. 7 and we can explain something from it. The performance on one MIC is almost twice as that on one CPU (61X). Because of good scalability of binomial option pricing, it can get approximate linear performance improvement on multi-CPU or multi-MIC. When it comes to MIC heterogeneous system, the speedup is 93X for one CPU and one MIC, which cannot reach ideal performance provided by one CPU and one MIC because of the overhead caused by synchronization. In the end, we get 254X speedup on one compute node of Tianhe-2 containing two CPUs and three MICs.

6 Conclusion

In this paper, we have proposed a parallel implementation of binomial option pricing based on Intel MIC heterogeneous system. We have deployed several optimizing strategies including optimizing compiler options, paralleling with OpenMP and rewrite the key codes with SIMD instructions in order to make the best use of performance of Intel Xeon CPU and Intel Xeon Phi coprocessor. We also present a auto-tuning cache blocking algorithm to deal with the high L1

cache miss rate problem. The experimental results indicates that, our solution achieves 33X speedup on one Intel Xeon CPU and 61X speedup on one Intel Xeon Phi and on a compute node of Tianhe-2 our solution can achieve over 250-fold speedup compared with original serial algorithm.

In the future, we plan to extend our work to multiple compute nodes and find some ways to reduce the negative impact by the problem of synchronization and communication between CPUs and MICs.

Acknowledgments. We thank the anonymous reviewers for their valuable comments. This work is supported financially by the National Hi-tech Research and Development Program of China under contracts 2012AA010902.

References

1. Intel MIC Architecture. <https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner>
2. The message passing interface (mpi) standard. <http://www.mcs.anl.gov/research/projects/mpi/>
3. The openmp api specification for parallel programming. <http://openmp.org>
4. Corden, M.J., Kreitzer, D.: Consistency of floating-point results using the intel compiler or why doesnt my application always give the same answer. Technical report, Intel Corporation, Software Solutions Group (2009)
5. Cox, J.C., Ross, S.A., Rubinstein, M.: Option pricing: a simplified approach. *J. Financ. Econ.* **7**(3), 229–263 (1979)
6. Dixon, M., Chong, J., Keutzer, K.: Acceleration of market value-at-risk estimation. In: Proceedings of the 2nd Workshop on High Performance Computational Finance, p. 5. ACM (2009)
7. Dongarra, J.: Visit to the National University for Defense Technology Changsha. University of Tennessee 199, China (2013). <http://www.netlib.org/utk/people/JackDongarra/PAPERS/tianhe-2-dongarra-report.pdf>
8. Gerbessiotis, A.V.: Architecture independent parallel binomial tree option price valuations. *Parallel Comput.* **30**(2), 301–316 (2004)
9. Horasanli, M.: A comparison of lattice based option pricing models on the rate of convergence. *Appl. Math. Comput.* **184**(2), 649–658 (2007)
10. Hull, J.C.: Options, Futures, and Other Derivatives. Pearson Education, India (2006)
11. Jeffers, J., Reinders, J.: Intel Xeon Phi Coprocessor High-Performance Programming. Newnes, Boston (2013)
12. Jin, Q., Thomas, D.B., Luk, W., Cope, B.: Exploring reconfigurable architectures for tree-based option pricing models. *ACM Trans. Reconfigurable Technol. Syst. (TRETS)* **2**(4), 2586–2600 (2009). Article No. 21
13. Kwok, Y.K.: Mathematical Models of Financial Derivatives. Springer Science & Business Media, Heidelberg (2008)
14. Malladi, R.K.: Using intel vtune performance analyzer events/ratios optimizing applications (2009)

15. Newburn, C.J., Dmitriev, S., Narayanaswamy, R., Wiegert, J., Murty, R., Chinchilla, F., Deodhar, R., McGuire, R.: Offload compiler runtime for the intel xeon phi coprocessor. In: Parallel and Distributed Processing Symposium Workshops & Ph.D. Forum (IPDPSW), 2013 IEEE 27th International, pp. 1213–1225. IEEE (2013)
16. Savage, J.E., Zubair, M.: Cache-optimal algorithms for option pricing. *ACM Trans. Math. Soft. (TOMS)* **37**(1), 1–30 (2010). Article No. 7
17. Thulasiram, R.K., Dondarenko, D.: Performance evaluation of parallel algorithms for pricing multidimensional financial derivatives. In: International Conference on Parallel Processing Workshops, 2002, Proceedings, pp. 306–313. IEEE (2002)
18. Zubair, M., Mukkamala, R.: High performance implementation of binomial option pricing. In: Gervasi, O., Murgante, B., Laganà, A., Taniar, D., Mun, Y., Gavrilova, M.L. (eds.) ICCSA 2008, Part I. LNCS, vol. 5072, pp. 852–866. Springer, Heidelberg (2008)

Algorithms and Architectures for Parallel Processing
15th International Conference, ICA3PP 2015,
Zhangjiajie, China, November 18-20, 2015,
Proceedings, Part III
Wang, G.; Zomaya, A.; Martinez Perez, G.; Li, K. (Eds.)
2015, LI, 807 p. 355 illus. in color., Softcover
ISBN: 978-3-319-27136-1