Scientific Programming 17 (2009) 295–308
DOI 10.3233/SPR-2009-0290
IOS Press

Evaluating multicore algorithms on the unified memory model

John E. Savage [a,*] and Mohammad Zubair [b]

[a] *Brown University, Providence, RI 02912, USA*

[b] *Old Dominion University, Norfolk, VA 23529, USA*

Abstract. One of the challenges to achieving good performance on multicore architectures is the effective utilization of the underlying memory hierarchy. While this is an issue for single-core architectures, it is a critical problem for multicore chips. In this paper, we formulate the unified multicore model (UMM) to help understand the fundamental limits on cache performance on these architectures. The UMM seamlessly handles different types of multiple-core processors with varying degrees of cache sharing at different levels. We demonstrate that our model can be used to study a variety of multicore architectures on a variety of applications. In particular, we use it to analyze an option pricing problem using the trinomial model and develop an algorithm for it that has near-optimal memory traffic between cache levels. We have implemented the algorithm on a two Quad-Core Intel Xeon 5310 1.6 GHz processors (8 cores). It achieves a peak performance of 19.5 GFLOPs, which is 38% of the theoretical peak of the multicore system. We demonstrate that our algorithm outperforms compiler-optimized and auto-parallelized code by a factor of up to 7.5.

Keywords: Option pricing, memory hierarchy, multicore chips

1. Introduction

Processors with multiple cores are being manufactured by a number of vendors including IBM, Sun, Intel, AMD and Tilera. At present most contain between 2 and 16 cores. However, a few contain as many as 64–80 cores. Plans exist to scale up chips to several hundred cores. Multicore processors are organized to share information across cores using fast buses or a switching network that limit the number of cores that can be accommodated. To scale processors to many cores, the trend is to organize the cores in a two-dimensional grid with a router embedded with each core.

Unfortunately, the software (including the compilers) to exploit these cores lags behind hardware development. To achieve high performance, applications need to be explicitly coded. One of the challenges to obtaining good performance is the effective utilization of the underlying memory hierarchy. To achieve good performance it is essential that algorithms be designed to maximize data locality so as to best exploit the hierarchical cache structure. While the efficient use of memory hierarchies is important in serial processors, it is doubly important in multicore architectures. Multicore processors have several levels of memory hierarchy. To obtain good performance it is necessary to design algorithms that minimize I/O traffic to slower memories in the hierarchy [24].

Recently, cache-oblivious algorithms have been proposed that are independent of cache parameters and are thus more portable across different architectures [9,17,37,43]. However, portability comes at a price. Yotov et. al. have experimentally demonstrated that even highly optimized cache-oblivious programs perform significantly worse than corresponding cache aware programs for dense linear algebra applications [47]. They point to two major reasons for this performance gap, ineffective utilization of the pipeline by cache oblivious algorithms and the inability to effectively hide memory latency by cache oblivious algorithms.

In the past serial memory hierarchies and parallel computing have been extensively explored, mostly independently but also in a limited way together, for shared memory processors [6,14,15,19,22,23,29]. Researchers have developed models for parallel computing starting with the PRAM [16]. The weakness of the PRAM model is that it ignores communication cost for moving data between processors. This is addressed by later models, for example, the LPRAM [5], BSP [42],

[*] Corresponding author. Tel.: +1 401 863 76 42; Fax: +1 401 863 76 57; E-mail: jes@cs.brown.edu.

LogP [13] and Postal models [8]. These models ignore the memory hierarchy, which is addressed by the Memory Hierarchy Game [32] and several parallel hierarchical models such as LogP-HMM [31], LogP-UMH [3,31], the Parallel Memory Hierarchy (PMH) model [2], and parallel versions [28,44–46] of the serial memory hierarchy models of Aggarwal et al [1,4]. Recently, there has been some effort to develop models for analyzing performance on more advanced architectures [10,11,20].

One major limitation of all earlier models is their inability to model multicore processors with varying degrees of sharing of caches at different levels. In these models sharing happens for all processors at the level of main memory or through a network via the processors. By contrast, a multicore architecture can have an L2 cache shared by a subset of cores, and an L3 cache by a larger subset of cores, and so on. (The Intel Dunnington processor has an L2 cache that is shared by two cores, and an L3 cache that is shared by all six cores.) In a multicore architecture the degree of cache sharing not only varies across cache levels, it varies from one architecture to another. For example, the Sun UltraSPARC T2 has an L2 cache that is shared by all eight cores as opposed to the Intel Dunnington processor that has an L2 cache that is shared by only two cores.

In addition, all earlier models lack a general strategy for deriving lower bounds to the communication traffic within and across cores. We need strong lower bounds not only to measure the effectiveness of proposed algorithms, but for insight in developing cache-efficient algorithms. Most of the efforts in deriving lower bounds are restricted to using strategies specific to applications and work for a limited set of architectures. See, for example, [5,27,45,46].

In this paper, we introduce the unified multicore model (UMM) that addresses all these limitations. It is an extension of the memory hierarchy game (MHG) developed for a single processor attached to a hierarchy of memories [32]. The model assumes that sets of cores share first-level caches, these share second-level caches, etc. and that the cache capacity is the same for all caches at a given level. The UMM seamlessly handles different types of multi-core processors with varying degrees of sharing of caches at different levels.

The proposed model works for computations that can be represented as directed acyclic graphs (DAGs). This includes matrix multiplication, FFT computation, and trinomial option pricing, which is discussed later. To derive lower bounds for a given DAG, we compute its S-span [32]. The S-span intuitively represents the maximum amount of computation that can be done after loading data in a cache at some level without accessing higher levels (those further away from the CPU) memories. A more precise definition of S-span is given later.

We demonstrate that the S-span of a DAG captures the computational dependencies inherent in the DAG and use it to develop lower bounds on communication traffic for a single core and multiple core architectures. The use of lower bounds in designing efficient multicore algorithms is an iterative process. We design a multicore algorithm and then analyze it on the model to determine the memory traffic at different levels of a memory hierarchy. We compare this performance with the derived lower bounds, and if the proposed algorithm is far away from the optimal we try to improve the algorithm and repeat this process. In absence of lower bounds, the algorithm designer does not have confidence in the efficiency of his/her code in terms of the number of memory references. This is particularly true of problems where efficient implementations are not known. The option pricing problem falls into this category.

Using our model, we derive lower bounds on memory traffic between different levels of hierarchy for financial and scientific computations. We also design and implement a multicore algorithm for option pricing using the trinomial model on a on a multicore system with two Quad-Core Intel Xeon 5310 1.6 GHz processors with a total of 8 cores. We analyze the proposed algorithm on our model and demonstrate that it exhibits a constant-factor optimal amount of memory traffic between different levels. When the algorithm was implemented, it outperformed the compiler-optimized and auto-parallelized code by a factor of up to 7.5.

The rest of the paper is organized as follows. Section 2 gives an overview of the unified multicore model and states the main theorem, which is then used in deriving lower bounds on memory traffic between different levels of a memory hierarchy. In Section 3 we describe the computational requirements for option pricing using the trinomial model. In Section 4 we discuss the optimal algorithm for valuing options using the trinomial model and its implementation including experimental results. In Section 5 we derive lower bounds on the amount of memory traffic between cache levels for three common scientific and financial problems. Finally, in Section 6 we draw conclusions.

2. Bounding memory traffic

In this section, we introduce the unified multicore model [34] and the multilevel memory hierarchy game and derive lower bounds on memory traffic between different levels of a memory hierarchy.

2.1. *Unified multicore model*

The unified multicore model (UMM), sketched in Fig. 1(a), captures the essential features of multicore cache hierarchies. It assumes that each core sees L levels of memory including Level-0, which refers to registers that are part of the core. It also assumes that all caches at a given level have the same size and that they are shared by the same number of cores. For our model, we define the following parameters for $1 \leqslant l \leqslant L-1$:

p_l: Number of cores sharing a cache at level-l.
α_l: Number of caches at the lth level.
σ_l: Size of a cache at level-l.

Observe that $p_{L-1} = p$, the total number of cores. Because the number of cores sharing a cache at a given level is the same for all caches at that level, it follows that $\alpha_l = p/p_l$. The highest level cache, main memory, is assumed to have unlimited capacity.

The model parameters for some sample multicore architectures [21,26,39–41] are given in Fig. 1(b).

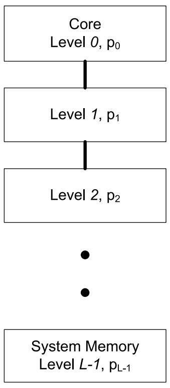
2.2. *The multicore memory hierarchy game*

The multicore memory hierarchy game (MMHG) is a pebbling game played on a DAG that models computations done on the UMM. It extends the memory hierarchy game (MHG) introduced in [32] and generalized in [33], p. 537. In Section 2.3 we improve upon previous lower bounds on the memory traffic required by the serial MHG. This is extended to the UMM in Section 2.4.

The rules of the MMHG are given below. The purpose of the game is to pebble the output vertices of a graph $G = (V, E)$. The value associated with a vertex is computed in a core register. This is modeled by placing a zero-level pebble on the vertex. A core cannot compute the value of an operator unless all its operands are present in core registers. This requirement is captured by requiring that to pebble a vertex with a zero-level pebble, all predecessors of that vertex also carry zero-level pebbles associated with the core. The number of zero-level pebbles available to a core is equal to its number of registers.

First-level pebbles correspond to locations in a first-level cache. There are as many pebbles as there are locations. When a register containing a value that must be retained spills over to a first-level cache, this is modeled by placing a first-level pebble on the vertex that corresponds to the value in the cache. Although in principle a value could be in both a register and a cache location, we generally assume that movement to a first-level cache corresponds to swapping zero- and first-level pebbles. Data in a first-level cache is available to all the processors sharing the cache, if any. Movement of data from a register to a first-level cache is one way that cores share information.

Caches at higher levels in a multicore hierarchy operate in the same way as a first-level cache. Data from a lower level cache spills over to the higher level cache and data in a higher level cache is available to all lower level caches and processors that share it. Sharing is the mechanism used to move data between cores. A level-l



(a)

Product	Levels	Cores	p_1	p_2	p_3	p_4
Sun UltraSPARC T2	4	8	1	8	8	
Intel Dunnington	5	6	1	2	6	6
Sun Rock	4	16	2	16	16	
Tilera Tile64	5	64	1	1	64	64
IBM Cell	3	8	1	1	8	

(b)

Fig. 1. (a) The unified multicore model and (b) model parameters of some sample multicore architectures.

pebble corresponds to a location in a level-l cache. Data moves up and down the hierarchy by swapping it between adjacent levels. Pebbles are associated with individual caches and model locations in those caches.

Note that the MMHG is a parallel pebbling game; pebbles associated with different caches can be placed and removed simultaneously, although this assumption is not strictly necessary.

We are interested in the communication traffic between adjacent levels in the hierarchy. This is modeled by the number of times that a level-l pebble is placed on a vertex containing a level-$(l-1)$ pebble or vice versa.

2.2.1. Rules of the MMHG

R1 (Computation step). A zero-level pebble associated with a core can be placed on any vertex all of whose immediate predecessors carry zero-level pebbles associated with that core.

R2 (Pebble deletion). Except for level-L pebbles on output vertices, a pebble at any level can be deleted from any vertex.

R3 (Initialization). A level-L pebble can be placed on an input vertex at any time.

R4 (Input from level-l). For $1 \leqslant l \leqslant L$, a level-$(l-1)$ pebble ξ associated with cache $c_{i,l-1}$ can be placed on any vertex carrying a level-l pebble associated with the parent cache of $c_{i,l-1}$.

R5 (Output to level-l). For $1 \leqslant l \leqslant L$, a level-$l$ pebble ξ associated with a cache $c_{i,l}$ can be placed on any vertex carrying a level-$(l-1)$ pebble associated with any cache $c_{j,l-1}$ that is a child of $c_{i,l}$.

R6 Each output vertex must be pebbled with a level-L pebble.

Let $\sigma_i^{(0)}$ be the number of registers in the ith core, $1 \leqslant i \leqslant p$. This is also the number of zero-level pebbles or size of the ith zero-level cache associated with the ith core. Let $\sigma_i^{(l)}$ be the number of level-l pebbles associated with the ith cache at level l, namely, $c_i^{(l)}$, $1 \leqslant i \leqslant \alpha_l$, where α_l is the number of caches at level l. In the UMM, $\sigma_i^{(l)} = \sigma_l$, a constant for all i. The number of cores sharing $c_i^{(l)}$ is $\tau_i^{(l)}$ where $\tau_i^{(l)} = p_l$ in the UMM. Finally, $\alpha_l p_l = p$ in the UMM where p is the total number of cores.

Although the MMHG is a parallel pebbling game, it can be serialized. That is, we can pebble one vertex at a time. This restriction does not alter the vertices at which I/O operations are performed, just the total time for the operations.

In memory hierarchies either the multilevel inclusion or exclusion policy is enforced. In the former, a copy of the value in each location in a level-l cache is maintained in all higher level caches. These copies may be dirty, that is, not currently consistent with the value in the lowest level cache containing the original, and are updated as needed. The exclusion policy, which applies to the above rules, does not reserve space for values held in lower level caches. The results are derived for this case. However, they also hold for the inclusion policy when the memory associated with a cache in the lower bounds is the difference between the capacity of a cache and that of all its subcaches.

The MHG is the variant of the MMHG in which there is only one processor and one cache at each of $L-1$ levels [32]. The level-L cache has unlimited size. We denote with $T_l^{(L)}(\Sigma, G)$ the number of I/O operations at level l on the DAG G where $\Sigma = (\sigma_1, \dots, \sigma_{L-1})$ denotes the sizes of the caches.

2.3. Uni-processor lower bounds

To set the stage for deriving lower bounds on communication traffic with the MMHG, we begin by describing the methods used to obtain lower bounds for the MHG. The lower bounds rely on the S-span measure of a graph G.

Definition. The S-span of a DAG G, $\rho(S, G)$, is the maximum number of vertices of G that can be pebbled in a zero-level pebble game starting with any initial placement of S zero-level pebbles.

$\rho(S, G)$ is most useful for graphs that have a regular structure. It provides good lower bounds on communication traffic for matrix multiplication, the fast Fourier transform, the pyramid graph and other graphs on the serial MHG.

The following gives a lower bound to $T_l^{(L)}(\Sigma, G)$, the number of I/O operations at level l in the MHG. The first version of this result appeared in [32]. This result improves upon the version in [33], p. 535, by tightening the lower bound when the number of memory locations below level-l is large.

Theorem 2.1. *Consider a pebbling of the DAG G with n input and m output vertices in an L-level memory hierarchy game. Let $\rho(S, G)$ be the S-span of G and $|V^*|$ be the number of vertices in G other than the inputs. Assume that $\rho(S, G)/S$ is a non-decreasing function of S.*

Then, for $0 \leqslant l \leqslant L-1$ the communication traffic between the lth and $(l-1)$st levels, $T_l^{(L)}(\Sigma, G)$, satisfies *the following lower bound where $\Sigma_{(l-1)} = \sum_{r=1}^{l-1} \sigma_r$ is the number of pebbles at all levels up to and including level-$(l-1)$.*

$$T_l^{(L)}(\Sigma, G) \geqslant \frac{\Sigma_{(l-1)}|V^*|}{\rho(2\Sigma_{(l-1)}, G)}.$$

It is also trivially true that $T_l^{(L)}(\Sigma, G) \geqslant (n+m)$.

Proof. The following is shown in [33], p. 535, where $T_2^{(2)}(S, G)$ is the number of I/O operations in the two-level MHG played with S zero-level pebbles.

$$T_l^{(L)}(\Sigma, G) \geqslant T_2^{(2)}(\Sigma_{(l-1)}, G).$$

This is derived by observing that placement of pebbles below (at or above) level l can be simulated with zero-level (first-level) pebbles. $T_2^{(2)}(\Sigma_{(l-1)}, G)$ is a lower bound because it provides the player with more freedom to place pebbles than the multi-level game.

The following inequality extends and simplifies the Hong–Kung lower bound on the number of I/O operations to pebble a graph [25]. It improves upon the bound of [33], p. 535. $|V^*|$ is the number of non-input vertices in G.

$$T_2^{(2)}(S, G) \geqslant \frac{S|V^*|}{\rho(2S, G)}.$$

To prove the above, partition a two-level pebbling strategy into h intervals, $h = \lceil T_2^{(2)}(S, G)/S \rceil$, such that each, except possibly the last, has S I/O operations, which has $S_1 \leqslant S$ I/O operations. ($h = 1$ when $S \geqslant T_2^{(2)}(S, G)$.) Thus,

$$T_2^{(2)}(S, G) = (h-1)S + S_1. \tag{1}$$

We now derive an upper bound on the number of vertices of G that are pebbled with computation steps within each interval and use this number to obtain a lower bound on h.

Consider one of the first $h-1$ intervals. Some vertices are pebbled with zero-level pebbles that do not have pebbles on them (computations are done on them). Others that carry zero-level pebbles are pebbled with first-level pebbles. (These are output operations.) Finally, some vertices that carry first-level pebbles are pebbled with zero-level pebbles. (These are input operations.)

The times at which computations, inputs, and outputs occur are generally intermixed making it difficult to overbound the number of computation steps. To simplify the analysis we provide an additional S zero-level pebbles so that if there are O output operations, we use O zero-level pebbles to allow such pebbles to stay on vertices until near the end of the interval. Similarly, if there are $I = S - O$ input operations, we use I zero-level pebbles to allow the input operations to occur near the beginning of the interval. Then, inputs occur at the beginning of the interval, outputs occur at the end, and computation steps occur in between. By the definition of S-span, the number of computation steps is at most $\rho(2S, G)$ because $2S$ zero-level pebbles are used.

The number of computation steps required is $|V^*|$, the number of non-input vertices. (At least one input operation must be performed on each input vertex.) The number of computation steps over the first $h-1$ intervals is at most $(h-1)\rho(2S, G)$. At most $\rho(2S_1, G)$ computation steps are performed in the last interval from which the following inequality holds.

$$\rho(2S_1, G) + (h-1)\rho(2S, G) \geqslant |V^*|. \tag{2}$$

Solving (2) for $(h-1)$ and substituting into (1), we have the following.

$$T_2^{(2)}(S, G) \geqslant \frac{S|V^*|}{\rho(2S, G)} + S_1\left(1 - \frac{\rho(2S_1, G)/2S_1}{\rho(2S, G)/2S}\right).$$

Because $S_1 \leqslant S$ and $\rho(S, G)/S$ is an increasing function of S,

$$T_2^{(2)}(S, G) \geqslant \frac{S|V^*|}{\rho(2S, G)}.$$

The lower bound $T_l^{(L)}(\Sigma, G) \geqslant (n+m)$ follows by observing that in a two-level MHG at least one input operation is required on each input vertex and at least one output operation on each output vertex. □

2.4. *Multicore lower bounds*

To extend the above results to the unified multicore model (UMM) we assume that the task of pebbling the vertices of a graph G with zero-level pebbles is shared among the cores and that no two cores perform the same computation. Pebbling is done according to the rules stated in Section 2.2.1. Each vertex of a graph G

is pebbled once with a zero-level pebble by some core. Associated with each core and each cache is a set of pebbles that are not interchangeable.

The number of I/O operations performed on a cache depends on the vertices of a graph G that are pebbled with zero-level pebbles by the cores sharing the cache. If these cores pebble very few (many) vertices, the number of I/O operations should be small (large).

Let $T_{i,l}(\Sigma, G)$ be the number of I/O operations for the ith cache at level-l in the hierarchy where $\Sigma = (\sigma_0, \sigma_1, \dots, \sigma_{L-1})$ is the list of storage capacities of the caches in the UMM. Let $T_{i,l}(\Sigma, G)$ be the number of I/O operations performed by the ith cache at level-l. We derive lower bounds to $T_{i,l}(\Sigma, G)$, one for the worst case in which do not know how the computational work (pebbling with zero-level pebbles) is distributed and a second when the work is uniformly distributed across all cores.

We generalize Theorem 2.1 to the UMM by reducing the problem of computing memory traffic between a cache and its subcaches or between a first-level cache and a core to the problem of computing the traffic between two levels in the serial MHG. We make three observations: (a) The kth core is responsible for pebbling with zero-level pebbles (computing) a subgraph $G_k = (V_k, E_k)$ of the graph $G = (V, E)$, (b) the MMHG can be serialized without changing the I/O to and from a cache, and (c) the traffic between a cache $c_i^{(l)}$ and its subcaches is the sum of the traffic to each subcache.

Let $\Gamma_{i,l}$ be the set of cores that have cache $c_i^{(l)}$ as their parent at level-l. Let $V_{i,l}$ be the vertices in the subgraphs G_k for $k \in \Gamma_{i,l}$. That is, these are the vertices that are pebbled by the cores that have cache $c_i^{(l)}$ as its level-l cache. Let $V_{i,l}^*$ be the non-input vertices in this set.

The following theorem derives lower bounds to the memory traffic under two conditions, (a) the worst case when it is not known that the workload is balanced between cores and (b) the case when the workload is uniformly distributed across all cores.

Theorem 2.2. *Consider a pebbling of the graph G in an L-level unified memory hierarchy game with p processors. Let $\rho(S, G)$ be the S-span of G and $|V^*|$ be the number of vertices in G other than the inputs. Assume that $\rho(S, G)/S$ is a non-decreasing function of S. Let β_{l-1} be the number of pebbles at level-$(l-1)$ and below in those caches having a cache at level-l as parent. Let α_l be the number of caches at level-l.*

For any allocation of workload to cores, for each l there is a level-l cache such that the communication traffic, $T_{l,M}^{(L)}(\Sigma, G)$, satisfies the following minimal lower bound.

$$T_{l,M}^{(L)}(\Sigma, G) \geqslant \frac{\beta_{l-1}(|V^*|/\alpha_l)}{\rho(2\beta_{l-1}, G)}.$$

When the workload is uniformly distributed over all cores, the communication traffic at a level-l cache, $T_{l,U}^{(L)}(\Sigma, G)$, satisfies the following bound. It is at least as strong as the above bound because $\rho(S, G)/S$ is a non-decreasing function of S and $\alpha_l/\alpha_{l-1} \leqslant 1$.

$$T_{l,U}^{(L)}(\Sigma, G) \geqslant \frac{\beta_{l-1}(\alpha_l/\alpha_{l-1})(|V^*|/\alpha_l)}{\rho(2\beta_{l-1}(\alpha_l/\alpha_{l-1}), G)}.$$

Proof. Consider first the worst-case result. Per the above discussion, a lower bound on the traffic between $c_i^{(l)}$ and its subcaches can be derived by observing that for some i, $|V_{i,l}^*| \geqslant |V^*|/\alpha_l$. This follows because there are α_l level-l caches and for at least one of them, its cores process a subgraph containing $|V^*|/\alpha_l$ non-input vertices. Second, the locations in all caches that have $c_i^{(l)}$ as parent is $\beta_{l-1} = \sum_{j=1}^{l-1}(\alpha_j/\alpha_l)\sigma_j$. Because the traffic between $c_i^{(l)}$ and its subcaches is the same if the MMHG is serialized, the first lower bound given in Theorem 2.1 applies from which the bound on $T_{l,M}^{(L)}(\Sigma, G)$ follows. This takes care of the worst-case result.

Consider next the case when the work is distributed uniformly, that is, when $|V_k^*| = |V^*|/p$. We can then bound the traffic between cache $c_i^{(l)}$ and one of its subcaches and multiply the result by the number of subcaches, namely, α_{l-1}/α_l. But the number of non-input vertices pebbled by the cores that have a cache at level-$(l-1)$ as parent is $p_{l-1}|V^*|/p = |V^*|/\alpha_{l-1}$. Also, the number of memory locations at levels $\leqslant l-1$ used for this computation is $\beta_{l-1}/(\alpha_{l-1}/\alpha_l)$. Using Theorem 2.1 the traffic between one subcache and its parent cache at level-l is at least

$$\frac{\beta_{l-1}(\alpha_l/\alpha_{l-1})(|V^*|/\alpha_{l-1})}{\rho(2\beta_{l-1}(\alpha_l/\alpha_{l-1}), G)}.$$

Multiplying the result by α_{l-1}/α_l we obtain the desired lower bound to $T_{l,U}^{(L)}(\Sigma, G)$. □

We now illustrate these results on a representative set of problems.

3. Option pricing using trinomial model

An option contract is a financial instrument that gives the right to its holder to buy or sell a financial asset at a specified price, referred to as the strike price, on or before the expiration date. The current asset price, volatility of the asset, strike price, expiration time, and prevailing risk-free interest rate determine the value of an option. Trinomial option valuation is one of the popular approaches that values an option using a discrete time model [12,30]. The trinomial option pricing computation is represented using the directed acyclic graph with in-degree 3 denoted $G^{(n)}_{\text{triop}}$ of depth n on $2n+1$ leaves shown in Fig. 2. We divide the time to expiration into n intervals and let the root be at the present time and the leaves at expiration times. We use $G^{(n)}_{\text{triop}}$ to determine the price of an option at the root node iteratively, starting from the leaf nodes. The trinomial model assumes that the price of an asset can go three ways: up, down, and remain unchanged.

We now describe the computational requirements for option pricing using the trinomial model. In particular, we describe the computation for pricing a put option contract that gives the right to its holder to sell an asset whose current price is Q at a strike price K with the expiration time T. We assume that the prevailing risk-free interest rate is r, and the volatility of the asset is ν. To illustrate the computation, we divide the expiration time into n intervals with each time interval $\mathrm{d}t = T/n$. For more details on these models, please refer to [12,30].

We index ith node at level j in Fig. 2 by (j,i) where $1 \leqslant j \leqslant n+1$ and $1 \leqslant i \leqslant 2n+1-2(j-1)$. Let c_i^j and q_i^j be the option price and asset price, respectively, at (j,i). The computation is initialized by defining $q_i^1 = Qd^n u^{i-1}$, where $u = \mathrm{e}^{\lambda\nu\sqrt{\mathrm{d}t}}$, $d = \mathrm{e}^{-\lambda\nu\sqrt{\mathrm{d}t}}$ and λ is a free parameter, and $c_i^1 = \mathit{MAX}(K - q_i^1, 0)$. Option and asset prices c_i^j and q_i^j are iteratively computed at higher levels using the following equations. Here, p_u, p_d and p_m, are pseudo-probabilities that prices go up, down and remain the same.

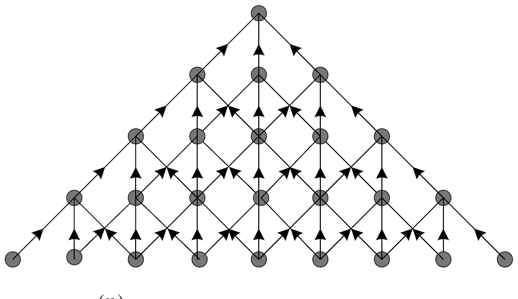


Fig. 2. $G^{(n)}_{\text{triop}}$ with depth $n = 4$ and $2n+1 = 9$ leaves.

$$
\begin{aligned}
c_i^{j+1} &= (p_u c_{i+2}^j + p_m c_{i+1}^j + p_d c_i^j)\mathrm{e}^{-r\,\mathrm{d}t},\\
q_i^{j+1} &= q_i^j * u,\\
c_i^{j+1} &= \mathit{MAX}(K - q_i^{j+1}, c_i^{j+1}),\\
p_u &= \frac{1}{2\lambda^2} + \frac{(r - \nu^2/2)\sqrt{\mathrm{d}t}}{2\lambda\nu},\\
p_m &= 1 - \frac{1}{\lambda^2}, \qquad p_d = \frac{1}{2\lambda^2} - \frac{(r-\nu^2/2)\sqrt{\mathrm{d}t}}{2\lambda\nu}.
\end{aligned}
$$

There are two types of options: European options, and American options. European options can only be exercised at the time of expiration, while American options can be exercised at any time prior to expiration. Note that computations of $q_i^{j+1} = q_i^j * u$ and $c_i^{j+1} = \mathit{MAX}(K - q_i^{j+1}, c_i^{j+1})$ are only required for American options. From the communication traffic perspective the difference between American and European options is that the American option requires access to an additional array that stores asset prices. The computation of a call option is similar except that the expression for the payoff c_i^{j+1} is replaced by $c_i^{j+1} = \mathit{MAX}(q_i^{j+1} - K, c_i^{j+1})$.

4. Multicore algorithm and implementation

In this section, we discuss the implementation on a multicore architecture of a representative application, option pricing. We propose and implement a multicore algorithm for a trinomial option pricing model. We implemented the proposed algorithms on a multicore system with two Quad-Cores Intel Xeon 5310 1.6 GHz processors for a total of 8 cores described in Fig. 3. A core has a 32 kB L1 data cache. The 4 mB L2 cache is shared by two cores. A single core of the Intel Xeon 5310 processor executes four floating-point instructions in one cycle, so the peak performance of a core is 6.4 GFLOPs with an overall peak of 51.2 GFLOPs for the complete system. In the UMM, $\alpha_1 = 8, \alpha_2 = 4, \alpha_3 = 1$. The sizes of caches in terms of the number of double-precision words holding values of c_i and q_i are $\sigma_1 = 2048$, and $\sigma_2 = 256$ kB.

To evaluate the performance of an algorithm, we use wall clock execution time. To evaluate how well a given algorithm matches the underlying architecture, we also compute algorithm performance as the per-

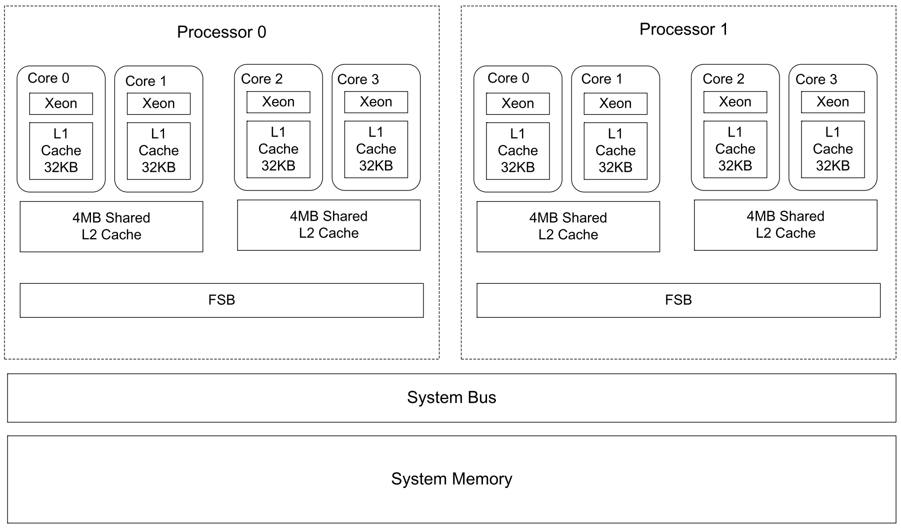


Fig. 3. A Dell PowerEdge 2990 system with two sockets and on each socket we have a Quad-Core Intel Xeon 5310 1.6 GHz processor.

centage of the theoretical peak performance for the target machine. For example, when we get 25.6 GFLOPs on 8 cores of our test system, our code is running at 50% of the peak. All our algorithms were compiled using Intel Visual Fortran Compiler 10.1 on Windows XP Professional Operating System. We compiled all our code with "-fast" option, which combines various complementary optimizations for the target processor.

4.1. Vanilla algorithm

Let us first look at issues in implementing a vanilla algorithm, which refers to a straightforward implementation of trinomial option pricing without any explicit partitioning for parallelism. A high-level description of the code is given in Algorithm 1. Note that the main computation is done inside the two nested loops (lines 17–19). The compiler faces two challenges to obtain good performance for the vanilla code: (a) effective utilization of the memory hierarchy; and (b) distributing the computation among different cores for concurrent execution.

It is easier to see the memory hierarchy issue when the code is being executed on a single core. The main data arrays, c_i and q_i, are accessed with a single stride in the innermost loop. Assume the L1 cache of the processor running a Vanilla algorithm can hold up to m elements of both the arrays c_i and q_i.

This implies that once we have accessed the first m elements of the arrays from the main memory, the L1 cache is full and cannot accommodate new data without replacing existing data. In other words, when we access the second set of elements of the array from main memory, it replaces the first set from the cache.

By the time we finish the first iteration of the outer loop, we have the last m elements of the array in cache. However, at the start of the second iteration of the outer loop, we again need to access the first m elements of the array, which unfortunately have been replaced from the cache. As a result, the processor has to go back to the main memory and get these elements. Thus, we are not reusing data in the cache, which results in a poor overall performance. The same issue exists in a multicore environment.

The second challenge for the compiler is to distribute computation among different cores along with efficient utilization of the memory hierarchy. Although compiler technology has made a lot of progress, it still cannot address some of these issues. The burden falls on the application programmer to partition a computation for effective utilization of the memory hierarchy and multiple cores.

For a multicore architecture, we need to partition the computation into blocks such that multiple cores can work concurrently on different blocks and at the same time effectively utilize the memory hierarchy. We propose one such partitioning that is illustrated in Fig. 4. For this partitioning, all blocks in a single row, for example blocks in jth row with labels $b_{j,*}$ can be executed concurrently. Observe that in our partitioning we have two types of square blocks. Alternate rows of blocks have the same type of blocks. For example, the second row and fourth row have the same type of blocks. We select a block size for this partitioning such

Algorithm 1 (VanillaTrinomial($Q, K, \mathrm{d}t, n, r, \lambda, \nu$))

```
1: $u \leftarrow \mathrm{e}^{\lambda\nu\sqrt{\mathrm{d}t}}$
2: $d \leftarrow \mathrm{e}^{-\lambda\nu\sqrt{\mathrm{d}t}}$
3: $p_u \leftarrow \frac{1}{2\lambda^2} + \frac{(r-\nu^2/2)\sqrt{\mathrm{d}t}}{2\lambda\nu}$
4: $p_m \leftarrow 1 - \frac{1}{\lambda^2}$
5: $p_d \leftarrow \frac{1}{2\lambda^2} - \frac{(r-\nu^2/2)\sqrt{\mathrm{d}t}}{2\lambda\nu}$
6: $\acute{p_u} \leftarrow p_u \mathrm{e}^{-r\,\mathrm{d}t}$
7: $\acute{p_m} \leftarrow p_u \mathrm{e}^{-r\,\mathrm{d}t}$
8: $\acute{p_d} \leftarrow p_d \mathrm{e}^{-r\,\mathrm{d}t}$
9: {initialization loop}
10: for $i = 1$ to $2n + 1$ do
11:     $q_i \leftarrow Qd^n u^{(i-1)}$ {$q_i$ is a 1-d array}
12:     $c_i \leftarrow MAX(K - q_i^1, 0)$ {$c_i$ is a 1-d array}
13: end for
14: {main computation loop}
15: for $j = 1$ to $n$ do
16:     for $i = 1$ to $2(n + 1 - j) - 1$ do
17:         $c_i \leftarrow \acute{p_u} c_{i+2} + \acute{p_m} c_{i+1} + \acute{p_d} c_i$
18:         $q_i \leftarrow q_i * u$
19:         $c_i \leftarrow MAX(K - q_i, c_i)$
20:     end for
21: end for
22: return $c_1$
```

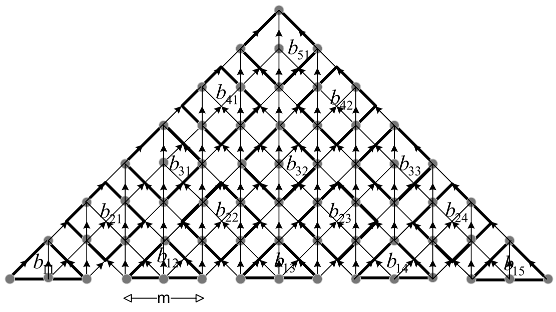


Fig. 4. Partitioning for a multicore architecture for depth $n = 7$, $2n + 1 = 15$ inputs, and block size $m = 3$.

that the required data for a block fits in the L1 cache of a core. Note that as we consider problem sizes up to a maximum of 64K leaf nodes, we can accommodate all the required data in the level-2 cache. Thus for our experimentation, we ignored partitioning for level-2. For the next level of memory, L0, which is the number of registers in the core, we rely on the compiler unrolling of the loop.

As shown in Fig. 5, processing of a block, $b_{j,i}$, requires:

- m north-east boundary output of block $b_{j-1,i}$,
- m north-west boundary output of block $b_{j-1,i+1}$,

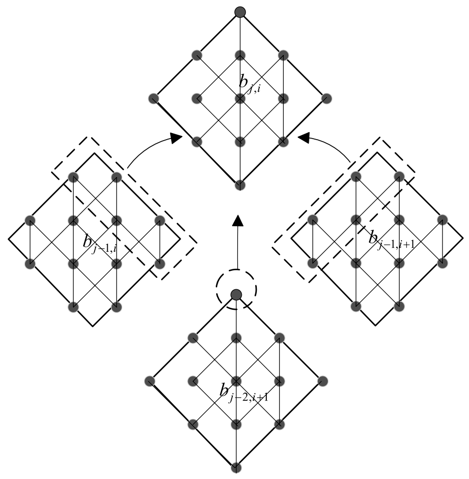


Fig. 5. Communication requirement for processing a block $b_{j,i}$.

- one north boundary output (top-most node) of block $b_{j-2,i+1}$.

Observe that blocks $b_{j-1,i}$ and $b_{j-1,i+1}$ are processed as part of a previous iteration; and block $b_{j-2,i+1}$ is processed two iterations in the past. A high-level description of the algorithm is given in Algorithm 2. Here we use *neb*($b_{j,i}$) to indicate the m north-east boundary elements of $b_{j,i}$ as shown in Fig. 6. Similarly, we define *nwb*($b_{j,i}$) to indicate the m north-west boundary elements of $b_{j,i}$. *top*($b_{j,i}$)) is the single data value of the block $b_{j,i}$ corresponding to the top-most node (see Fig. 6).

In trinomial partitioning, the processing of squares alternates between the two types as indicated in the algorithm. To keep our presentation simple, we ignore processing of the first row of blocks, which is similar to other rows except that a block is an incomplete square and it does not require input from an earlier processed block. We also assume that m evenly divides $2n + 1$; and $nb = (2n + 1)/m$ is an odd integer. If these assumptions are not correct, the run times are changed by small constant factors.

The output of neighbor $b_{j-1,i}$, north-east boundary, that is required for processing $b_{j,i}$ is stored as part of the shared array that holds the option prices. We do in-place computation in the shared array as we move from one level to next, similar to the vanilla algorithm. The output of the neighbor $b_{j-1,i+1}$, north-west boundary that is required for processing of $b_{j,i}$ is stored separately from the shared array. The same is true for the *top*($b_{j-2,i+1}$). To minimize the storage requirement, we reuse the array that stores north-west boundary of $b_{j-1,i+1}$ to store north-east boundary of $b_{j,i}$.

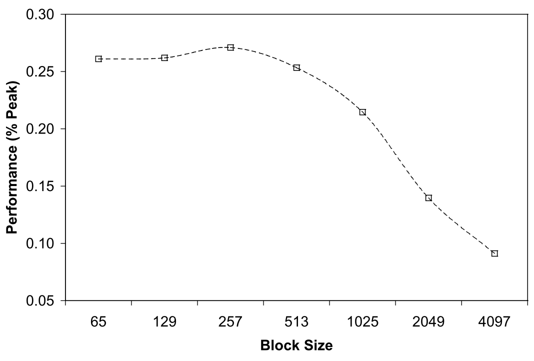


Fig. 6. Execution time as a function of block size.

Algorithm 2 ($G^{(n)}_{\text{triop}}$)

```
1:  for $j = 1$ to $nb - 1$ do
2:      {Process the following loop only for odd $j$s}
3:      {OpenMP directive is placed at this point of the code}
4:      for $i = 1$ to $nb - j$ do
5:          processSquareI($b_{j,i}$, $neb(b_{j-1,i})$,
6:                  $nwb(b_{j-1,i+1})$, $top(b_{j-2,i+1})$)
7:      end for
8:      {Process the following loop only for even $j$s}
9:      {OpenMP directive is placed at this point of the code}
10:     for $i = 1$ to $nb - j$ do
11:         processSquareII($b_{j,i}$, $neb(b_{j-1,i})$,
12:                 $nwb(b_{j-1,i+1})$, $top(b_{j-2,i+1})$)
13:     end for
14: end for
```

Observe that a large block size for L1 results in an unbalanced load distribution among cores. For a given problem size, there is an optimal block size for L1 as seen from the plots of Fig. 6. We use OpenMP directives to parallelize the computation across different cores. We use the work-sharing directive of OpenMP to distribute the iterations of the inner loop of Algorithm 2 among different cores using eight threads, one for each core.

The OpenMP directives are placed just before the second loop (line 3/line 9). Observe that processing of a block $b_{j,i}$ requires input from blocks that were processed in previous iterations $j - 1$ and $j - 2$. Hence the blocks in the current iteration can be concurrently executed. The workload for each thread (core) is decided by the schedule directive of OpenMP. We experimented with different schedules and found that the static schedule with chunk size of one gave optimal performance.

In [36] we have obtained the following upper bound on the S-span of $G^{(n)}_{\text{triop}}$.

Theorem 4.1. *The S-span of $G^{(n)}_{\text{triop}}$ satisfies $\rho(S, G^{(n)}_{\text{triop}}) \leqslant (S-1)^2/4$.*

Analysis

We first estimate the memory traffic between an L2 cache and one of its child L1 caches and then multiply it by two to get the estimate for $T_2^{(4)}$. Observe that our partitioning results in $nb(nb+1)/2$ blocks, where $nb = (2n+1)/m$. When n is large compared to the block sizes and the number of cores, the number of blocks is large and they are almost uniformly distributed among the various cores. The size of a block is approximately $2m^2$. Because there are eight cores, the number of blocks allocated to a core is approximately $n^2/8m^2$. Observe that processing of a typical block requires $2m+1$ data values. In the worst case we assume these values are not available in L1 cache. The estimate for memory traffic between one L2 cache and one of its child L1 caches is given by

$$T_2^{(4)}/2 \approx 2m\left(\frac{n^2}{8m^2}\right),$$

$$T_2^{(4)} \approx \frac{n^2}{4m}.$$

Using the balanced workload case of Theorems 2.2 and 4.1, the lower bound for $T_2^{(4)}$ is given approximately by

$$T_2^{(4)} \geqslant \frac{\alpha_1 n^2}{2\alpha_2^2 \beta_1}.$$

For our system $\alpha_1 = 8$, $\alpha_2 = 4$ and for $\beta_1 = 2m$ we have the following

$$T_2^{(4)} \geqslant \frac{n^2}{8m}.$$

Thus the proposed algorithm performance is bounded by a constant factor of 4 away from the lower bound. Observe that we assume the L1 cache holds m words or that $\sigma_1 = \beta_1/2 = m$. For our system, $\sigma_1 = 2048$ data values. Considering load balancing issues as discussed earlier, when $m = 2048$ and the problem size is small, the performance of the algorithm can be far from optimal. This is due to an artifact of our lower bounds, which ignore load balancing issues. It may be possible to strengthen our bounds by considering load balancing issues. Observe that for our system when we have a large L2 cache, we do not have a strong bound for $T_3^{(4)}$, which is trivially bounded by the number of inputs. Hence we ignore $T_3^{(4)}$ from our analysis.

Performance

We summarize our results in Tables 1–4. Our algorithm performs better for large problem sizes. We achieve 38% of the peak performance for a 66K problem size versus 27% of the peak performance for 8K problem size on 8 cores. For a 66K size problem, we obtained 19.4 GFLOPs. Similarly we observed a better scalability for large problem sizes. For example, we obtained a speedup of 7.4 for an 66K size problem on 8 cores versus a speedup of 5.7 for an 8K size problem on 8 cores.

For comparison, we also implemented a vanilla algorithm, which refers to a straightforward implementation of trinomial option pricing without any explicit partitioning for parallelism. We compiled the vanilla code with the "-fast" option along with the "-Qparallel" option that enables the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel. Our results for the vanilla algorithm are summarized in Table 5. The best performance for the vanilla algorithm is 6% of the peak as compared to 38% of the peak for our algorithm.

We note that previously proposed parallel implementations for pricing options suffer from low performance [18,38] as they do not address memory hierarchy issues. For example, Gerbessiotis' algorithm [18] only gets 1.5% of the peak performance on a single node of a Pentium cluster for $n = 32$K; and degrades to 0.8% on a 16-node cluster due to the communication overheads.

Table 1
Execution time for the trinomial algorithm using OpenMP

	Execution time (s)			
$2n + 1$	1 core	2 cores	4 cores	8 cores
8481	0.0588	0.0309	0.0171	0.0103
16705	0.2259	0.1160	0.0613	0.0342
33345	0.8500	0.4339	0.2266	0.1232
66177	3.3399	1.6870	0.8633	0.4519

Table 2
Performance of the trinomial algorithm as percentage of theoretical peak

	Peak (%)			
$2n + 1$	1 core	2 cores	4 cores	8 cores
8481	38.2	36.4	32.8	27.2
16705	38.6	37.6	35.5	31.9
33345	40.9	40.0	38.3	35.3
66177	41.0	40.6	39.6	37.9

Table 3
Performance of the trinomial algorithm in GFLOPs

	GFLOPS			
$2n + 1$	1 core	2 cores	4 cores	8 cores
8481	2.4	4.7	8.4	13.9
16705	2.5	4.8	9.1	16.3
33345	2.6	5.1	9.8	18.1
66177	2.6	5.2	10.1	19.4

Table 4
Scalability performance of the trinomial algorithm

	Speed up			
$2n + 1$	1 core	2 cores	4 cores	8 cores
8192	1.0	1.9	3.4	5.7
16384	1.0	1.9	3.7	6.6
32768	1.0	2.0	3.8	6.9
65536	1.0	2.0	3.9	7.4

Table 5
Performance of the vanilla trinomial algorithm when using "-fast" and "-Qparallel" compiler options for optimization and auto-parallelization

$2n + 1$	Execution time (s)	GFLOPS	Peak (%)
8481	0.05	3.07	6.0
16705	0.22	2.55	5.0
33345	0.86	2.59	5.1
66177	3.41	2.57	5.0

5. Applications of the methodology

The bounds on I/O traffic for multicore chips derived here can be applied to problems characterized by DAGs. All that is needed are the parameters of the UMM for the chip being analyzed and the S-span of the graphs in question. We illustrate this for matrix multiplication, the fast Fourier transform (FFT), and other financial computations such as the binomial pricing model (BPM) that is characterized by the graph shown in Fig. 7.

Matrix multiplication

We consider any straight-line program for the multiplication of two $n \times n$ matrices A and B that perform the same set of additions and multiplications as the standard algorithm but in an arbitrary order. Each computation is described by a DAG G_{MM}. Let $C = AB$ be the result matrix.

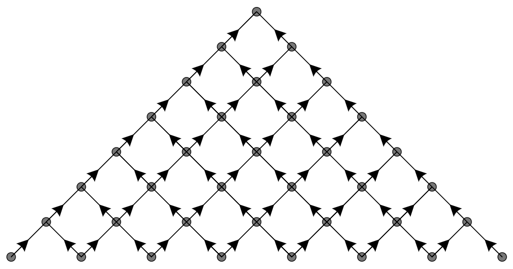


Fig. 7. The graph $G^{(n)}_{\text{biop}}$ with depth n and $n+1=8$ leaves.

Theorem 5.1 [33]. *The S-span of any $n \times n$ matrix multiplication DAG G_{MM} satisfies $\rho(S, G_{\text{MM}}) \leqslant 2S^{3/2}$.*

Applying Theorem 2.2 to G_{MM} we have the following result.

Theorem 5.2. *When the workload in computing the $n \times n$ matrix multiplication graph is uniformly distributed across all cores, each level-l cache in the UMM requires a number of I/O operations satisfying the following bound where α_l is the number of caches at level l and β_l is the number of storage locations of caches that have a level-l cache as parent.*

$$T^{(L)}_{l,U}(\Sigma, G_{\text{MM}}) \geqslant \frac{\sqrt{\alpha_{l-1}}\, n^2(2n-1)}{2\sqrt{2}\alpha_l^{3/2}\sqrt{\beta_{l-1}}}.$$

Also, $T^{(L)}_{l,U}(\Sigma, G_{\text{MM}}) \geqslant \alpha_{l-1}^{1/3}((n-1)n^2)^{2/3}/(2^{5/3}\alpha_l)$.

Proof. The first result follows from the fact that $|V^*| = n^2(2n-1)$ for G_{MM} and the bound $\rho(S, G_{\text{MM}}) \leqslant 2S^{3/2}$. The second lower bound follows from the fact that S_0 satisfies $\rho(2S_0, G) \geqslant |V^*|/\alpha_{l-1}$. □

The fast Fourier transform algorithm (FFT)

The FFT graph is well known. A bound on its S-span is given below.

Theorem 5.3 [7]. *The S-span of the n-input FFT graph satisfies $\rho(S, G_{\text{FFT}}) \leqslant 2S \log_2 S$.*

Applying Theorem 2.2 to G_{FFT} we have the following result.

Theorem 5.4. *When the workload in computing the n-input fast Fourier transform graph is uniformly distributed across all cores, each level-l cache in the UMM requires a number of I/O operations satisfying the following bound where α_l is the number of caches at level-l and σ_l is the number of storage locations in a level-l cache. and β_l is the number of storage locations of caches that have a level-l cache as parent.*

$$T^{(L)}_{l,U}(\Sigma, G_{\text{FFT}}) \geqslant \frac{n \log_2 n}{4\alpha_l \log_2(2\beta_{l-1}(\alpha_l/\alpha_{l-1}))}.$$

Also, $T^{(L)}_{l,U}(\Sigma, G_{\text{FFT}}) \geqslant (n \log_2 n)/((4\alpha_{l-1}) \times (\log_2(n \log_2 n) - \log_2(2\alpha_{l-1})))$.

Proof. To obtain the first bound note that $|V^*| = n \log_2 n$ and $\rho(S, G_{\text{FFT}}) \leqslant 2S \log_2 S$. The second bound requires the smallest value of S_0 satisfying $\rho(2S_0, G) \geqslant |V^*|/\alpha_{l-1}$. A relaxed condition is that $(2S_0)\log_2(2S_0) \geqslant a = (n \log_2 n)/(2\alpha_{l-1})$. Substitution shows that if $a \geqslant 2$, then $2S_0 \geqslant a/\log_2 a$. In this case, $S_0 \geqslant (n \log_2 n)/((4\alpha_{l-1})(\log_2(n \log_2 n) - \log_2(2\alpha_{l-1})))$. Multiplying by α_{l-1}/α_l, we have the desired result. □

The binomial pricing model

The binomial pricing model for options is similar to that for the trinomial pricing model. The difference is that it does not assume that prices may remain constant on a time step. It is described by the graph $G^{(n)}_{\text{biop}}$ shown in Fig. 7.

Theorem 5.5 [36]. *The S-span of $G^{(n)}_{\text{biop}}$ satisfies $\rho(S, G^{(n)}_{\text{biop}}) \leqslant S(S-1)/2$.*

Applying Theorem 2.2 to $G^{(n)}_{\text{biop}}$ we have the following result.

Theorem 5.6 [35]. *When the workload in computing the binomial graph $G^{(n)}_{\text{biop}}$ on $n+1$ inputs is uniformly distributed across all cores, each level-l cache in the UMM requires a number of I/O operations satisfying the following bound where α_l is the number of caches at level-l and β_{l-1} is the number of storage locations in all caches that have a given level-l cache as parent.*

$$T^{(L)}_{l,U}(\Sigma, G^{(n)}_{\text{biop}}) \geqslant \frac{\alpha_{l-1} n(n+1)}{4\alpha_l^2 \beta_{l-1}}.$$

Also, $T^{(L)}_{l,U}(\Sigma, G^{(n)}_{\text{biop}}) \geqslant \sqrt{\alpha_{l-1}}\, n/2\alpha_l$.

6. Conclusions

In this paper we introduce a new model for the cache hierarchy of multicore chips. We also study cache-efficient algorithms for these chips. Our results are illustrated by applying them to option pricing, a compute-intensive problem. Cache-efficiency is examined by developing lower bounds on the memory traffic between levels of a cache hierarchy and applying them to a variety of problems including the trinomial model for option pricing. For the latter we not only exhibit an algorithm that is theoretically efficient, we show it gives excellent performance on a 8-core chip, a platform with two Quad-Core Intel Xeon 5310 1.6 GHz processors. Our algorithm achieves a peak performance of 19.5 GFLOPs, which is 38% of the theoretical peak of the multicore system. It outperforms the compiler optimized and auto-parallelized code by a factor of up to 7.4 on a problem with 66K vertices.

Acknowledgment

The work by John E. Savage was supported in part by NSF Grant CCF-0403674.

References

[1] A. Aggarwal, B. Alpern, A.K. Chandra and M. Snir, A model for hierarchical memory, in: *STOC'87: Proceedings of 19th Annual ACM Symposium on the Theory of Computing*, New York, 1987, pp. 305–314.

[2] B. Alpern, L. Carter and J. Ferrante, Modeling parallel computers as memory hierarchies, in: *Proceedings of the 1993 Conference on Programming Models for Massively Parallel Computers*, Berlin, Germany, 1993, pp. 116–123.

[3] B. Alpern, L. Carter, E. Feig and T. Selker, The uniform memory hierarchy model of computation, *Algorithmica* **12**(2,3) (1994), 72–109.

[4] A. Aggarwal, A.K. Chandra and M. Snir, Hierarchical memory with block transfer, in: *FOCS'87: 28th Annual Symposium on Foundations of Computer Science*, Los Angeles, CA, October 1987, pp. 204–216.

[5] A. Aggarwal, A.K. Chandra and M. Snir, Communication complexity of PRAMs, *Theor. Comput. Sci.* **71**(1) (1990), 3–28.

[6] R.C. Agarwal, F.G. Gustavson and M. Zubair, Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms, *IBM J. Res. Dev.* **38**(5) (1994), 563–576.

[7] A. Aggarwal and J.S. Vitter, The input/output complexity of sorting and related problems, *Commun. ACM* **31**(9) (1988), 1116–1127.

[8] A. Bar-Noy and S. Kipnis, Designing broadcasting algorithms in the Postal model for message-passing systems, *Math. Syst. Theory* **27**(5) (1994), 431–452.

[9] M.A. Bender, E.D. Demaine and M. Farach-Colton, Cache-oblivious *B*-trees, in: *FOCS'00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, 2000, p. 399.

[10] G. Bilardi, K. Ekanadham and P. Pattnaik, Optimal organizations for pipelined hierarchical memories, in: *SPAA'02: Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, Winnipeg, MB, Canada, 2002, pp. 109–116.

[11] G.E. Blelloch, R.A. Chowdhury, P.B. Gibbons, V. Ramachandran, S. Chen and M. Kozuch, Provably good multicore cache performance for divide-and-conquer algorithms, in: *SODA'08: Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, CA, 2008, pp. 501–510.

[12] J.C. Cox, S.A. Ross and M. Rubinstein, Option pricing: A simplified approach, *J. Financ. Econ.* **7**(3) (1979), 229–263.

[13] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian and T. von Eicken, LogP: towards a realistic model of parallel computation, *SIGPLAN Notices* **28**(7) (1993), 1–12.

[14] J.J. Dongarra, J. DuCroz, S.Hammarling and R. Hanson, Algorithm 656: An extended set of Fortran basic linear algebra subprograms: Model implementation and test programs, *ACM T. Math. Software* **14** (1988), 18–32.

[15] J.J. Dongarra, J. DuCroz, S. Hammarling and R. Hanson, An extended set of Fortran basic linear algebra subprograms, *ACM T. Math. Software* **14** (1988), 1–17.

[16] S. Fortune and J. Wyllie, Parallelism in random access machines, in: *STOC'78: Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, San Diego, CA, 1978, pp. 114–118.

[17] M. Frigo, C.E. Leiserson, H. Prokop and S. Ramachandran. Cache-oblivious algorithms, in: *FOCS'99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, 1999, p. 285.

[18] A.V. Gerbessiotis, Architecture independent parallel binomial tree option price valuations, *Parallel Comput.* **30**(2) (2004), 301–316.

[19] K. Goto and R.A. van de Geijn, Anatomy of high-performance matrix multiplication, *ACM T. Math. Software* **34**(3) (2008), 1–25.

[20] N.K. Govindaraju, S. Larsen, J. Gray and D. Manocha, A memory model for scientific algorithms on graphics processors, in: *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, Tampa, FL, ACM, New York, NY, 2006, p. 89.

[21] M. Gschwind, H.P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe and T. Yamazaki, Synergistic processing in cell's multicore architecture, *IEEE Micro* **26**(2) (2006), 10–24.

[22] A. Gupta, F.G. Gustavson, M. Joshi and S. Toledo, The design, implementation, and evaluation of a symmetric banded linear solver for distributed-memory parallel computers, *ACM T. Math. Software* **24**(1) (1998), 74–101.

[23] F.G. Gustavson, High-performance linear algebra algorithms using new generalized data structures for matrices, *IBM J. Res. Dev.* **47**(1) (2003), 31–55.

[24] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, CA, 2007.

[25] J.-W. Hong and H.T. Kung, I/O complexity: The red-blue pebble game, in: *STOC'81: Proceedings 13th Annual ACM Symposium on Theory of Computing*, 1981, pp. 326–333.

[26] Intel's multicore architecture briefing, http://www.intel.com/pressroom/archive/releases/20080317fact.htm, 2008.

[27] D. Irony, S. Toledo and A. Tiskin, Communication lower bounds for distributed-memory matrix multiplication, *J. Parallel Distrib. Comput.* **64**(9) (2004), 1017–1026.

[28] B.H.H. Juurlink and H.A.G. Wijshoff, The parallel hierarchical memory model, in: *SWAT'94: Proceedings of the 4th Scandinavian Workshop on Algorithm Theory*, Springer, New York, NY, 1994, pp. 240–251.

[29] B. Kågström, P. Ling and C. van Loan, GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark, *ACM T. Math. Software* **24**(3) (1998), 268–302.

[30] Y.K. Kwok, *Mathematical Models of Financial Derivatives*, Springer, Singapore, 1998.

[31] Z. Li, P.H. Mills and J.H. Reif, Models and resource metrics for parallel and distributed computation, in: *HICSS'95: Proceedings of the 28th Hawaii International Conference on System Sciences*, 1995, p. 51.

[32] J.E. Savage, Extending the Hong–Kung model to memory hierarchies, in: *Computing and Combinatorics*, D.-Z. Du and M. Li, eds, LNCS, Vol. 959, Springer, 1995, pp. 270–281.

[33] J.E. Savage, *Models of Computation: Exploring the Power of Computing*, Addison–Wesley, Reading, MA, 1998.

[34] J.E. Savage and M. Zubair, Memory hierarchy issues in multicore architectures, Technical Report CS-08-08, Department of Computer Science, Brown University, 2008.

[35] J.E. Savage and M. Zubair, A unified model for multicore architectures, in: *IFMT'08: ACM/IEEE Proceedings of the First International Forum on Next-Generation Multicore/Manycore Technologies*, November 2008, pp. 1–12.

[36] J.E. Savage and M. Zubair, Cache-optimal algorithms for option pricing, *ACM T. Math. Software* (2009), to appear.

[37] S. Sen, S. Chatterjee and N. Dumir, Towards a theory of cache-efficient algorithms, *J. ACM* **49**(6) (2002), 828–858.

[38] R.K. Thulasiram and D.A. Bondarenko, Performance evaluation of parallel algorithms for pricing multidimensional financial derivatives, in: *ICPPW'02: Proceedings of the International Conference on Parallel Processing Workshops*, 2002, pp. 306–313.

[39] Tile64 processor family, available at: http://www.tilera.com/products/processors.php, 2008.

[40] M. Tremblay and S. Chaudhry, A third-generation 65 nm 16-core 32-thread plus 32-scout-thread CMT SPARC© processor, available at: http://blogs.sun.com/HPC/resource/RockISSCC08.pdf, 2008.

[41] UltraSPARC T2 Processor – Overview, available at: http://www.sun.com/processors/UltraSPARC-T2/, 2008.

[42] L.G. Valiant, A bridging model for parallel computation, *Commun. ACM* **33**(8) (1990), 103–111.

[43] L.G. Valiant, A bridging model for multi-core computing, in: *ESA'08: Proceedings 16th Annual European Symposium on Algorithms*, LMCS, Vol. 5193, Springer, Berlin, 2008, pp. 13–28.

[44] J.S. Vitter and E.A.M. Shriver, Optimal disk I/O with parallel block transfer, in: *STOC'90: Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, Baltimore, MD, 1990, pp. 159–169.

[45] J.S. Vitter and E.A.M. Shriver, Algorithms for parallel memory II: hierarchical multilevel memories, *Algorithmica* **12**(2,3) (1994), 148–169.

[46] J.S. Vitter and E.A.M. Shriver, Algorithms for parallel memory I: two-level memories, *Algorithmica* **12**(2,3) (1994), 110–147.

[47] K. Yotov, T. Roeder, K. Pingali, J. Gunnels and F. Gustavson, An experimental comparison of cache-oblivious and cache-conscious programs, in: *SPAA'07: Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, San Diego, CA, 2007, pp. 93–104.