A Unified Model for Multicore Architectures*

John E. Savage
Brown University
Providence,Rhode Island 02912
jes@cs.brown.edu

Mohammad Zubair
Old Dominion University
Norfolk, Virginia 23529
zubair@cs.odu.edu

ABSTRACT

With the advent of multicore and many core architectures, we are facing a problem that is new to parallel computing, namely, the management of hierarchical parallel caches. One major limitation of all earlier models is their inability to model multicore processors with varying degrees of sharing of caches at different levels. We propose a unified memory hierarchy model that addresses these limitations and is an extension of the MHG model developed for a single processor with multi-memory hierarchy. We demonstrate that our unified framework can be applied to a number of multicore architectures for a variety of applications. In particular, we derive lower bounds on memory traffic between different levels in the hierarchy for financial and scientific computations. We also give a multicore algorithms for a financial application that exhibits a constant-factor optimal amount of memory traffic between different cache levels. We implemented the algorithm on a multicore system with two Quad-Core Intel Xeon 5310 1.6GHz processors having a total of 8 cores. Our algorithms outperform compiler optimized and auto-parallelized code by a factor of up to 7.3.

Categories and Subject Descriptors

G.4 [**Mathematical Software**]: Efficiency, Algorithm design and analysis

General Terms

Algorithms, Performance, Experimentation

Keywords

Memory Hierarchy, Multicore

1. INTRODUCTION

*This paper appears in the Proceedings of IFMT'08, the First International Forum on Next-Generation Multicore/Manycore Technologies, Cairo, Egypt, November 24-25, 2008.

Power consumption and heating constraints are limiting the instruction-level parallelism for improving processor performance. The response of the industry has been to increase the number of cores on a die. The effective use of these cores requires the parallelization of applications. Parallelization is a problem that arose in a serious way in the 1980s and 1990s. The use of multiple cores has introduced a new problem to parallel computing, namely, the management of hierarchical parallel caches.

Multicore chips employ a cache structure to simulate a fast common memory. Unfortunately, the software to exploit these caches is lagging behind hardware development. To achieve high performance, applications executed on multicore chips need to be explicitly coded. 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 on these processors it is necessary to design algorithms that minimize I/O traffic to slower memories in the hierarchy [22]. Researchers have extensively studied the problem of designing efficient algorithms for processors with memory hierarchy [15, 16, 26, 20, 18, 5, 21].

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 to 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.

One salient characteristics of multicore architectures is that they have a varying degrees of sharing of caches at different levels. Most architectures have cores with a private L1 cache. Depending on the architecture, an L2 cache is shared by two or more cores and an L3 cache is shared by four or more cores. The main memory is typically shared by all cores. The degree of sharing at a level varies from one multicore processor to another. In this paper, we explore these architectures from the memory hierarchy perspective. We propose a unified memory hierarchy model that captures

essential features of each of these architecture.

Researchers in the past have explored various models for parallel computing starting with the PRAM [17]. 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 [8], BSP [34], LogP [14], and Postal models [12]. These models ignored the memory hierarchy, which is addressed by the Memory Hierarchy Game [29] and several parallel hierarchical models such as LogP-HMM [28], LogP-UMH [10, 28], the Parallel Memory Hierarchy (PMH) model [11], and parallel versions [35, 37, 36, 25] of the serial memory hierarchy models of Aggarwal *et al* [6, 7].

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 we not only have varying degrees of sharing of caches at different levels, the degree of sharing at a level varies from one multicore 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 that can help in deriving lower bounds for communication traffic within a core and across cores for different applications. Most of the efforts in deriving lower bounds are restricted to using strategies specific to an application and they work for a limited set of architectures. See for example [8, 37, 36, 24].

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 [29]. 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 multiple-core processors with varying degrees of sharing of caches at different levels.

We also provide a framework for deriving lower bounds on communication traffic between different levels of memory in the UMM. These bounds are useful to implementers for not only designing efficient algorithms, but also to see limitations of the architectures in achieving optimal performance. For example, the bounds can help to determine that the bandwidth available on the bus that interconnects main memory to caches at upper levels is not sufficient to achieve optimal performance.

The proposed model works for straight-line computations all of which can be represented as directed acyclic graphs (DAGs). This includes matrix multiplication, FFT computation, and binomial option pricing, for example. To derive lower bounds for a given DAG, we first compute its S-span [29]. 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. Our model and associated analysis help in designing efficient multicore algorithms. We demonstrate that our unified framework can be applied to a variety of multicore architectures for a variety of applications. In particular, we derive lower bounds on memory traffic between different levels of hierarchy for financial and scientific computations. We also give a multicore algorithms for a financial application that exhibits a constant-factor optimal amount of memory traffic between different levels. We implemented these algorithms on a multicore system with two Quad-Core Intel Xeon 5310 1.6GHz processors with a total of 8 cores. We demonstrate that our algorithms outperform compiler-optimized and auto-parallelized code by a factor of up to 7.5.

In Section 2 we describe a number of commercial multicore chips. This leads to the formulation of the UMM in Section 3. In Section 4 we model computations on the UMM as a multi-level pebbling game. General lower bounds on the communication traffic between caches in the UMM are derived in Section 6. In Section 7 we apply these bounds to three problems, a method for pricing options as well as matrix multiplication and the FFT algorithm. In Section 8 an implemention for a method of pricing options is described and analyzed. Conclusions are stated in Section 9.

2. MULTICORE ARCHITECTURES

In this section, we describe the memory hierarchies associated with a number of important multicore architectures. Most of the architectures have a private L1 cache. Depending on the architecture, the L2 and L3 caches are typically shared by two or more cores.

Sun UltraSPARC T2

The UltraSPARC T2 processor [4] has eight SPARC processor cores. A core has an 8-Kbyte, 4-way data cache. Each is connected through a crossbar to a 4 Mbyte, 16-way L2 cache. The L2 cache is banked eight ways to support eight cores and connects to four on-chip DRAM controllers, which directly interface to a pair of fully buffered DIMM (FBD) channels. (See Figure 1).

Intel Six-Core Dunnington Processor

The Intel Dunnington processor [1] consists of six cores with each core having its own 96KB L1 data cache. The 3-MB L2 cache is shared by two cores. The 16-MB L3 cache is shared by all six cores. The interface to main memory is via a front side bus that operates at 1066 MT/seconds. (See Figure 2).

Sun Rock Processor with 16 Cores

The Rock processor [33] is the latest 16-core SPARC system from Sun. It was scheduled to be released in 2008 but is now expected to be released in 2009. The Sun Rock processor

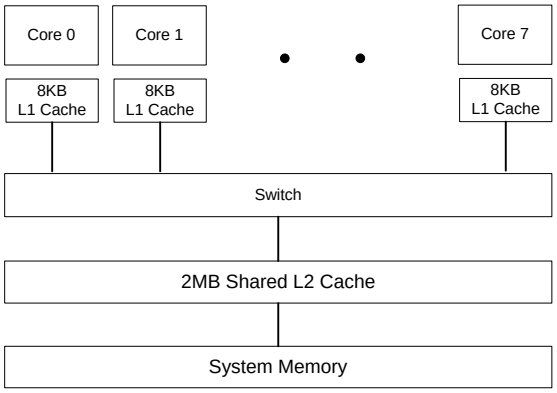

Figure 1: The 8-Core Sun UltraSPARC T2 Processor

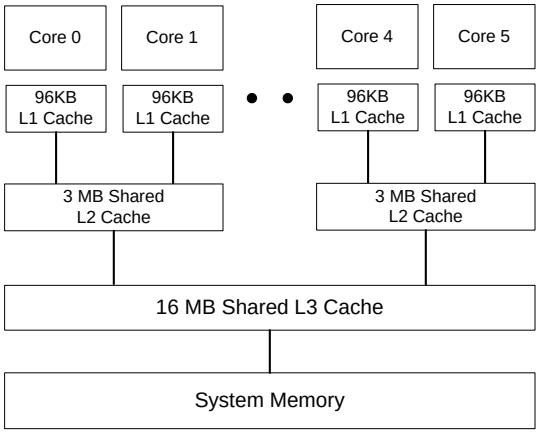

Figure 2: The Intel 6-Core Dunnington Processor

consists of four clusters of four cores each, for a total of 16 cores. There is a 32KB dual banked 4-way L1 data cache that is shared by two cores. A 2MB 4-bank 8-way L2 cache is shared by all the cores. The memory hierarchy used by the Sun Rock Processor is illustrated in Figure 3.

Tilera 64-core Tile64 and Intel 80-core Tera-Scale Processors

The Tile64 processor [3] consists of an 8×8 grid of processor cores. The Tile64 architecture eliminates on bus chip interconnect, which limits the number of cores that can be put on a chip. Each processor core has a communication switch that connects it to a two dimensional on-chip mesh network called the iMesh. A processor core has an 8KB L1 data cache and a 64 KB L2 cache. The architecture treats all the L2 caches together as a single large L3 cache. Multicore coherent caching enables data cached on a core to be accessed by other cores using the iMesh network. The block diagram illustrating the memory hierarchy for a single core is given in Figure 4. Intel is experimenting with a large number of cores in its Tera-Scale 80-core processor [2]. As with the Tilera processor, it connects cores in a 2D mesh.

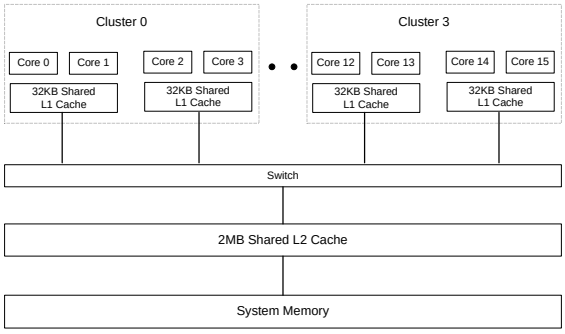

Figure 3: The Sun 16-Core Rock Processor

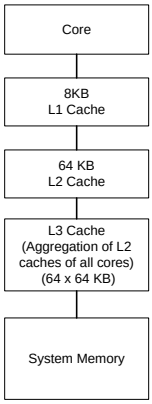

Figure 4: Memory Hierarchy for a Single Core of a 64-Core Tile64 Processor from Tilera

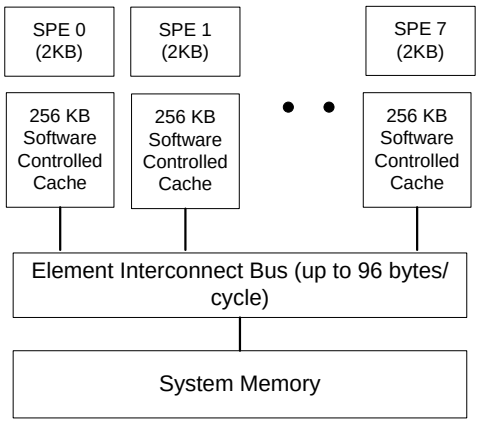

Figure 5: Memory Hierarchy for SPEs in a IBM, Sony, and Toshiba Cell Broadband Engine

IBM, Sony, and Toshiba Cell Broadband Engine Multicore Architecture

IBM, Sony, and Toshiba have developed a multicore chip (the cell broadband engine) [19] that consists of eight synergistic processor elements (SPEs) and a PowerPC Processor Element (PPE). The SPE design is optimized for computation-intensive applciations and the PPE is optimized for control tasks. The architecture for the SPE does not support a traditional cache-based memory hierarchy, although it does have a local store associated with each element under direct program control. The local store can be viewed as a software controlled cache where the data is moved between local store and memory using DMA. This simplifies the memory hierarchy hardware and moves the complexity to the programmer/compiler for managing local store. An SPE has a register file consisting of 128 registers of 128-bits each (2 KB of storage), and a local store of 256 KB. (See Figure 5).

3. UNIVERSAL MULTICORE MODEL

In this section we introduce the **universal multicore model (UMM)** (see Figure 6) that captures the essential features of the multicore cache hierarchies described above. It assumes that each core sees L levels of memory including Level-0, which refers to registers and are typically part of the core. It also assumes that all caches at a 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 $1 \leq l \leq L-1$.

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 $\alpha_l = p/p_l$. Parameters of the chips in Section 2 are given in Table 1.

4. THE MULTICORE MEMORY HIERARCHY GAME

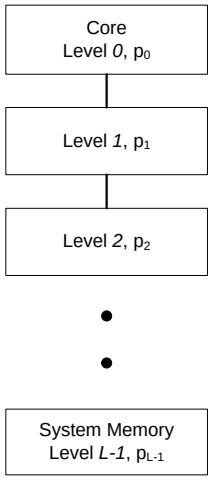

Figure 6: The Universal Multicore Model in which p_l denotes the number of cores sharing a common cache.

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

Table 1: The parameters of the multicore chips described herein.

In this section we introduce the **multicore memory hierarchy game (MMHG)**, a pebbling game played on a DAG that models computations done on the UMM. It is an extension of the memory hierarchy game (MHG) introduced in [29] and generalized in [30, p. 535]. In Section 5 we improve upon previous lower bounds on the memory traffic required by the serial MHG. In Section 6 we extend this analysis to the UMM.

The MMHG assumes that there are p cores and an L-level hierarchy of caches. Each level-l cache, $1 \leq l \leq L-1$, is shared by p_l cores. The MMHG assumes that all caches at level l have capacity σ_l and that the highest level cache or main memory has unlimited capacity. The sizes of caches at the various levels is represented by the tuple $\Sigma = (\sigma_1, \ldots, \sigma_{L-1})$. It isn't necessary to specify the size of the highest level memory because it is unlimited. The rules of the MMHG are given below. The purpose of the game is to pebble the output vertices of a graph $G = (V, E)$.

It is important to emphasize that the MMHG is a parallel pebbling game; pebbles associated with different caches can be placed and removed simultaneously. It is helpful to think of pebble moves as occuring synchronously, 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.

4.1 Rules of the MMHG

We now state the rules for the multicore memory hierarchy game (MMHG). These rules generalize those for the memory hierarchy game (MHG) [29] that models data movement up and down a cache hierarchy for uni-processors. As mentioned above, in the MMHG multiple pebble moves can occur simultaneously as long as they don't violate the rules.

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

R5. (Output to Level-l) For $2 \leq l \leq L$, 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.

Rule R1 states that a value associated with a vertex can be computed in a core only if the data on which the value depends are present in the core. Zero-level pebbles can slide from a predecessor to a successor vertex, which corresponds to using a register as both the source and target of an operation. The second rule states that data can be discarded. Rule R3 says that input data is available in the highest level cache and can be freely accessed. The fourth and fifth rules govern input from and output to level l. The fourth rule allows data associated with a vertex to move to level l from level $l-1$ by placing a level-$(l-1)$ pebble on a vertex carrying a level-l pebble. Of course, this data movement is only possible from a cache to one of those caches to which it is connected. The fifth rule works in the same way except that it refers to movement from a lower level to the next higher one. The last rule states the goal of the pebble game, namely, to place a level-L pebble on each output vertex. This is tantamount to storing computed results in a permanent memory.

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 policy 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. They are updated as needed. The exclusion policy does not reserve space for values held in lower level caches. The rules given above are written for the exclusion policy. The results derived below are for the exclusion case. However, the results also hold for the inclusion policy when the memory associated with a cache in the lower bounds is the amount of memory over and above that used to hold copies of values in lower level caches.

The MHG is the variant of the MMHG in which there is only one processor and one cache at each of $L-1$ levels. 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 $\Sigma = (\sigma_1, \ldots, \sigma_{L-1})$ denotes the sizes of the caches.

5. 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 otain lower bounds for the MHG. The lower bounds rely on the S-span measure of a graph G.

DEFINITION 1. *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 red pebbles.*

The S-span is a measure of how many vertices can be pebbled without doing any I/O. S pebbles are placed on the most fortuitous vertices of a graph and the maximum number of vertices that can be pebbled without doing I/O is the value of the S-span. Clearly, the measure is most useful for

graphs that have a fairly regular structure. It has provided good lower bounds on communication traffic for matrix multiplication, the Fast Fourier Transform, the pyramid graph and other graphs. This definition applies even if a DAG G is not a connected graph.

The following theorem derives a lower bound to $T_l^{(L)}(\Sigma, G)$, the number of I/O operations at level l in the MHG. It is a generalization to hierarchical memories of a result of Hong and Kung [23]. The first version of this theorem appeared in [29]. The result given here improves upon the version given in [30, p. 535] by tightening the lower bound when the number of memory locations below level l is large. A proof of this theorem can be found in [32].

THEOREM 5.1. *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 $1 \leq l \leq L-1$ 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 $l-1$.

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

$T_l^{(L)}(\Sigma, G)$ also satisfies $T_l^{(L)}(\Sigma, G) \geq S_0$ where S_0 is the smallest integer satisfying $\rho(2S_0, G) \geq |V^*|$. It is also trivially true that $T_l^{(L)}(\Sigma, G) \geq (n+m)$.

6. MULTICORE LOWER BOUNDS

We now extend the above results to the uniform 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. As with the serial model of computation, we assume that each vertex of a graph G is pebbled once with a zero-level pebble by some core. Pebbles are local to either core registers or higher level caches.

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 $\Sigma = (\sigma_1, \sigma_2, \ldots, \sigma_{L-1})$ is the list of storage capacities of the caches in the UMM. We derive a lower bound to $T_{i,l}(\Sigma, G)$ for the ith cache at level l whose cores pebble the largest number of vertices of G with zero-level pebbles.

We generalize Theorem 5.1 to the UMM. Recall that the multicore memory hierarchy game (MMHG) is a parallel pebbling game in which it is possible to move multiple pebbles at the same time. We establish the following result by simulating the parallel pebbling of a graph in the UMM by a serial pebbling of the graph with two levels of pebbles.

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

THEOREM 6.1. *Consider a pebbling of the graph G in an L-level universal 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.

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

Also, $T_{l,M}^{(L)}(\Sigma, G) \geq S_0$ where S_0 is the smallest value satisfying the bound $\rho(2S_0, G) \geq |V^*|/\alpha_l$.

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 $\alpha_l/\alpha_{l-1} \leq 1$.

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

Also, $T_{l,U}^{(L)}(\Sigma, G) \geq (\alpha_{l-1}/\alpha_l)S_0$ where S_0 is the smallest integer such that $\rho(2S_0, G) \geq |V^*|/\alpha_{l-1}$.

PROOF. The proof uses Theorem 5.1 for uni-processors. It has two parts. The first part consists of a lower bound to the I/O at level l in terms of the number of I/O operations in a two-level game. The important storage parameter is β_{l-1}, the number of memory locations in all the caches below level l that have a common level-l cache as parent. The second part consists of a lower bound on the I/O complexity for the two-level serial game.

We extend the first result to the ith cache at level l in the UMM, denoted $c_{i,l}$. Let $G_{i,l}$ denote the subgraph of the graph G that is computed by the cores that have this cache as a parent. Recall that no zero-level vertex pebblings are done by more than one core.

In the proof of Theorem 5.1 we argue that with two types of pebble we can simulate the restriction on the MHG. One type of pebble is used to simulate pebbles below level l. The second is used to simulate pebbles at or above level l. If we segment these pebbles into pebbles of the proper number at each of the levels of the hierarchy, the number of I/O operations at the level-l cache is that determined by the rules of the MHG. If, however, we now drop these restrictions and treat them as two types of pebble, we loosen the restrictions and derive a potentially smaller bound on the I/O at the level-l cache.

We apply the same type of reasoning here. We also argue that from the point of view of I/O operations at cache $c_{i,l}$, there is no loss in assuming that the pebblings at levels $\leq l-1$ within the cores and caches that have $c_{i,l}$ as a parent are serial. The same is true for pebblings at higher levels or in other caches and cores. We use one type of pebble for levels $\leq l-1$ and a second type for levels $\geq l$. The number of the second type of pebble is unlimited.

Now consider pebbling $G_{i,l}$. We first derive a lower bound that holds for any allocation of work to cores. Let β_{l-1} be the total number of pebbles associated with all caches that share cache $c_{i,l}$ as a parent. Since there are α_j/α_l caches of capacity σ_j at level j that have $c_{i,l}$ as parent, $\beta_{l-1} = \sum_{j=1}^{l-1}(\alpha_j/\alpha_l)\sigma_j$. Consequently, we see that the number of I/O operations at cache $c_{i,l}$ satisfies the following lower bound.

$$T_{i,l}^{(L)}(\Sigma, G) \geq T_2^{(2)}(\beta_{l-1}, G_{i,l})$$

We have reduced the number of I/O operations at a particular cache to the number of I/O operations in the serial two-level game. It remains to derive a lower bound for this number of I/O operations. We observe that in the UMM each cache $c_{i,l}$ is associated with a subgraph $G_{i,l}$. Since there are α_l level-l caches, for some i, $G_{i,l}$ has $|V^*|/\alpha_l$ non-input vertices. Now we use the lower bound given in Theorem 5.1 to $T_2^{(2)}(S, G_{i,l})$

$$T_2^{(2)}(\beta_{l-1}, G_{i,l}) \geq \frac{\beta_{l-1}(|V^*|/\alpha_l)}{\rho(2\beta_{l-1}, G)}$$

from which the first lower bound follows.

From Theorem 5.1 we also have that $T_2^{(2)}(\beta_{l-1}, G_{i,l}) \geq S_0$ where S_0 is the smallest integer such that $\rho(2S_0, G) \geq |V_{i,l}^*|$ where the latter is the number of non-input vertices in $G_{i,l}$. Since $|V_{i,l}^*| \geq |V^*|/\alpha_l$, the second result follows.

To derive the third result, which holds when the work is allocated uniformly to cores, we derive a lower bound to the traffic between a single level-$(l-1)$ cache, say, $c_{j,l-1}$ and its parent, say $c_{i,l}$. We multiply this traffic by α_{l-1}/α_l, the number of level-$(l-1)$ caches with $c_{i,l}$ as parent. The storage capacity of $c_{j,l-1}$ and those caches for which it is a parent is $\beta_{l-1}^* = \beta_{l-1}(\alpha_l/\alpha_{l-1})$.

Let $G_{j,l-1}$ be the subgraph computed by the cores that have $c_{j,l-1}$ as parent. Then, a lower bound to the traffic between $c_{j,l-1}$ and $c_{i,l}$ is given below. Since $G_{j,l-1}$ has $|V^*|/\alpha_{l-1}$ non-input vertices, we have the following.

$$T_2^{(2)}(S, G_{j,l-1}) \geq \frac{\alpha_{l-1}}{\alpha_l} \frac{\beta_{l-1}^*(|V^*|/\alpha_{l-1})}{\rho(2\beta_{l-1}^*, G)}$$

For the last result we have that $T_2^{(2)}(\beta_{l-1}, G_{j,l-1}) \geq S_0$ where S_0 is the smallest integer such that $\rho(2S_0, G) \geq |V_{j,l-1}^*| \geq |V^*|/\alpha_{l-1}$. The traffic between all level-$(l-1)$ caches and a single level-l caches is $(\alpha_{l-1}/\alpha_l)S_0$. □

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

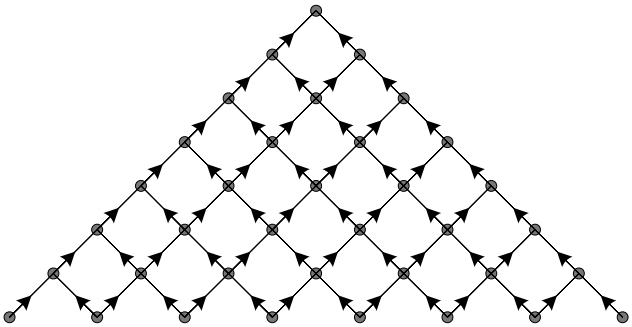

Figure 7: The graph $G_{biop}^{(n)}$ with depth n and $n+1 = 8$ leaves.

7. BOUNDS FOR SPECIFIC PROBLEMS

In this section, we investigate some common computations from financial and scientific domains and derive lower bounds on the number of I/O operations using the methodology given above. In the financial domain, we examine the option pricing computation using the binomial model. In the scientific domain, we study matrix multiplication and FFT computations.

7.1 Financial Computations

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 as the strike price, on or before the expiration date. Binomial option valuation is one popular discrete time methods of assigning a value to an option [27, 13].

The binomial option pricing computation is modelled by the directed acyclic pyramid graph $G_{biop}^{(n)}$ of depth n and $n+1$ leaves shown in Figure 7. Here the expiration time is divided into n intervals (defined by $n+1$ endpoints), the root is at the present time, and the leaves are at expiration times. We use $G_{biop}^{(n)}$ to determine the price of an option at the root node iteratively, starting from the leaf nodes.

$G_{biop}^{(n)}$ models a computation with time of duration $dt = T/n$. In $G_{biop}^{(n)}$ the level increases as we go up the tree. We identify i^{th} node at level j by (j, i), where $1 \leq j \leq n+1$ and $1 \leq i \leq n+2-j$. As part of initialization we define asset and option prices at leaf nodes ($j = 1$). Asset price q_i^1 at node $(1, i)$ is given by $q_i^1 = Qd^n u^{(i-1)}$, where $u = e^{\nu\sqrt{dt}}$ and $d = u^{-1}$. Here u and d indicate the fraction by which asset can go up or down respectively in one time interval. The initial price of the option at node $(1, i)$, c_i^1, is simply the option payoff at the node, which is given by $c_i^1 = MAX(K - q_i^1, 0)$ where K is the strike price. Next we iteratively compute option prices at nodes at level $j+1$ using prices at level j as defined below.

$$c_i^{j+1} = (p_u c_{i+1}^j + p_d c_i^j)e^{-rdt} \quad (1)$$

$$q_i^{j+1} = q_i^j * u \quad (2)$$

$$c_i^{j+1} = MAX(K - q_i^{j+1}, c_i^{j+1}) \quad (3)$$

Here, c_i^j and $q_i^j = Qd^n u^{2(i-1)+j-1}$ are the option price and asset price respectively at (j, i). Also, p_u and p_d are pseudo-probabilities given by

$$\begin{aligned} p_u &= \frac{e^{rdt} - d}{u - d} \\ p_d &= 1 - p_u \end{aligned}$$

The final output, c_1^{n+1} is the option price at the root node. 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 computation (2) and (3) are only required for American options. From the communication traffic perspective the difference between American and European option is that American option requires access to an additional array that stores asset prices. The computation for a call option is similar except that the expression for payoff (3) is replaced by the following.

$$c_i^{j+1} = MAX(q_i^{j+1} - K, c_i^{j+1})$$

In [31] we have obtained the following upper bound on the S-Span of $G_{biop}^{(n)}$.

THEOREM 7.1. *The S-Span of $G_{biop}^{(n)}$ satisfies $\rho(S, G_{biop}^{(n)}) \leq S(S-1)/2$.*

Applying Theorem 6.1 to $G_{biop}^{(n)}$ we have the following result.

THEOREM 7.2. *When the workload in computing the binomial graph 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.*

$$T_{l,U}^{(L)}(\Sigma, G_{biop}^{(n)}) \geq \frac{\alpha_{l-1} n(n+1)}{4\alpha_l^2 \beta_{l-1}}$$

Also, $T_{l,U}^{(L)}(\Sigma, G_{biop}^{(n)}) \geq \sqrt{\alpha_{l-1}} n/2\alpha_l$.

PROOF. The lower bound uses the fact that for all caches $c_{i,l}$, $1 \leq i \leq \alpha_l$, the number of non-input vertices in the subgraph pebbled by the cores that have $c_{i,l}$ as a parent is $|V^*|/\alpha_l$. The value of $|V^*|$ is $n(n+1)/2$ for $G_{biop}^{(n)}$. Using $\rho(S, G_{biop}^{(n)}) \leq S(S-1)/2 \leq S^2/2$ we have the lower bound. The second lower bound is obtained by multiplying S_0 by α_{l-1}/α_l where $\rho(2S_0, G_{biop}) \geq |V^*|/\alpha_l$. Using $\rho(S, G_{biop}^{(n)}) \leq S^2/2$ and replacing $|V^*|$ by the lower bound $n^2/2$, the result follows. □

7.2 Scientific Computations

Matrix Multiplication

We consider straight-line programs for the multiplication of two square matrices that perform the same set of additions and multiplications as the standard algorithm but in an arbitrary order. Such computations are described by DAGs. The result of multiplying two $n \times n$ matrices A and B is the matrix $C = AB$.

The S-span of matrix multiplication is given in [30, p. 541].

THEOREM 7.3. *The S-Span of $n \times n$ matrix muliplication satisfies $\rho(S, G_{MM}) \leq 2S^{3/2}$.*

Applying Theorem 6.1 to G_{MM} we have the following result. This result, in asymptotic form, was derived by Hong and Kung [23]. A concrete lower bound is due to Savage [29], [30, p. 542]. Using essentially the same basic proof technique, Irony *et al* [24] also derive a concrete lower bound but solely in the context of matrix multiplication.

THEOREM 7.4. *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.*

$$T_{l,U}^{(L)}(\Sigma, G_{MM}) \geq \frac{\sqrt{\alpha_{l-1}} n^2(2n-1)}{2\sqrt{2}\alpha_l^{3/2}\sqrt{\beta_{l-1}}}$$

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

PROOF. The lower bound uses the fact that for all caches $c_{i,l}$, $1 \leq i \leq \alpha_l$, the number of non-input vertices in the subgraph pebbled by the cores that have $c_{i,l}$ as a parent is $|V^*|/\alpha_l$. The value of $|V^*|$ is $n^2(2n-1)$ for G_{MM}. Using $\rho(S, G_{MM}) \leq 2S^{3/2}$ we have the lower bound. The second lower bound requires the smallest value of S_0 satisfying $\rho(2S_0, G) \geq |V^*|/\alpha_{l-1}$. Since $\rho(S, G) \leq 2S^{3/2}$ and $|V^*| \geq 2(n-1)n^2$, it follows that $S_0 \geq ((n-1)n^2)^{2/3}/(2^{5/3}\alpha_{l-1}^{2/3})$ from which the conclusion follows. □

The Fast Fourier Transform Algorithm

The complex Fourier Transform maps a tuple of complex coefficients $\boldsymbol{a} = (a_0, a_1, \ldots, a_{n-1})$ to a set of n values $\boldsymbol{b} = (b_0, b_1, \ldots, b_{n-1})$ by evaluating the polynomial $p(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$ at the roots of unity, that is, values in $\{e^{i2\pi j/n} \mid 0 \leq j \leq n-1\}$ where $b_j = p(e^{i2\pi j/n})$ and i is the solution to $x^2 = -1$.

The Fast Fourier Transform algorithm (FFT) is obtained by first writing $p(x) = p_e(x^2) + xp_o(x^2)$ where $p_e(x^2)$ ($xp_o(x^2)$) is the polynomial consisting of the coefficients of $p(x)$ of even (odd) degree. When the same decomposition is applied to $p_e(y)$ and $p_o(y)$ and their subpolynomials, we obtain the FFT algorithm. It is represented by the butterfly graph.

The S-span of the FFT graph is implicit in the the work of Hong and Kung [23]; a simplified proof the S-span is due to Aggrawal and Vitter [9]. See also [30, p. 546].

THEOREM 7.5. *The S-Span of the n-input FFT graph satisfies* $\rho(S, G_{FFT}) \leq 2S \log_2 S$.

Applying Theorem 6.1 to G_{FFT} we have the following result.

THEOREM 7.6. *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* β_l *is the number of storage locations of caches that have a level-l cache as parent.*

$$T_{l,U}^{(L)}(\Sigma, G_{FFT}) \geq \frac{n \log_2 n}{4\alpha_l \log_2(2\beta_{l-1}(\alpha_l/\alpha_{l-1}))}$$

Also, $T_{l,U}^{(L)}(\Sigma, G_{FFT}) \geq \frac{(n \log_2 n)}{(4\alpha_{l-1})(\log_2(n \log_2 n) - \log_2(2\alpha_{l-1}))}$.

PROOF. The lower bound uses the fact that for all caches $c_{i,l}$, $1 \leq i \leq \alpha_l$, the number of non-input vertices in the subgraph pebbled by the cores that have $c_{i,l}$ as a parent is $|V^*|/\alpha_l$. The value of $|V^*|$ is $n \log_2 n$ for G_{FFT}. Using $\rho(S, G_{FFT}) \leq 2S \log_2 S$, the lower bound follows. The second lower bound requires the smallest value of S_0 satisfying $\rho(2S_0, G) \geq |V^*|/\alpha_{l-1}$. Since $\rho(S, G) \leq 2S \log_2 S$, it follows that S_0 satisfies $(2S_0) \log_2(2S_0) \geq a = (n \log_2 n)/(2\alpha_{l-1})$. Straightforward substitution shows that if $a \geq 2$, then $2S_0 \geq a/\log_2 a$. In this case,

$S_0 \geq (n \log_2 n)/((4\alpha_{l-1})(\log_2(n \log_2 n) - \log_2(2\alpha_{l-1})))$.

Multiplying by α_{l-1}/α_l, we have the desired result. □

8. 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 binomial option pricing model. We implemented the proposed algorithms on a multicore system with two Quad-Cores Intel Xeon 5310 1.6GHz processors for a total of 8 cores described in Figure 8. A core has a 32KB L1 data cache. The 4MB 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 $\sigma_2 = 256KB$.

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 percentage of the theoretical peak performance in gigaflops 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.

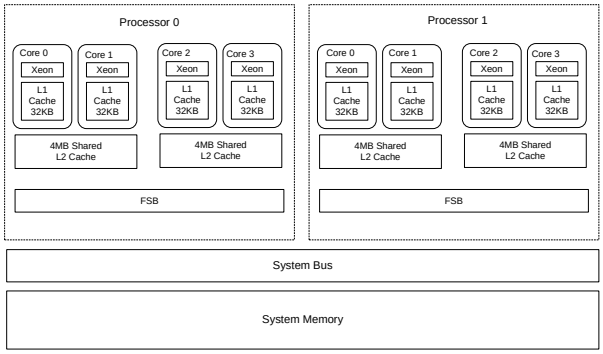

Figure 8: A Dell PowerEdge 2990 system with two sockets and on each socket we have a Quad-Core Intel Xeon 5310 1.6GHz processor.

8.1 Vanilla Algorithm

Let us first look at issues in implementing a vanilla algorithm, which refers to a straightforward implementation of binomial 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 15-17). 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 amongst different cores for concurrent execution. 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.

Algorithm 1 VanillaBinomial(Q, K, dt, n, r, ν)

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

8.2 Multicore Algorithm

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 heirarchy. We propose one such partitioning that is illustrated in Figure 9. For this partitioning,

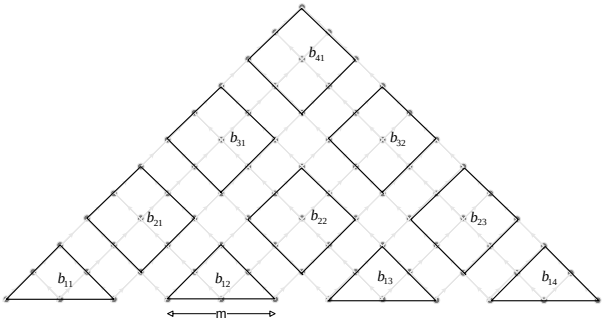

Figure 9: Partitioning for a multicore architecture for $n = 11$ and block size $m = 3$.

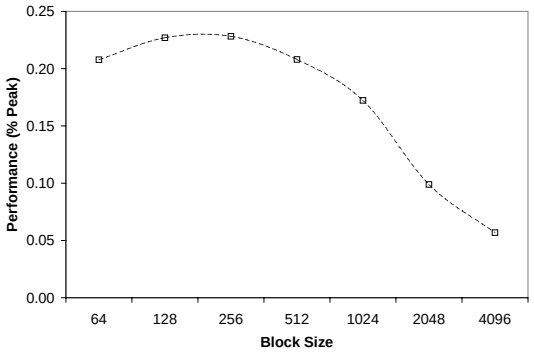

Figure 10: Execution time as a function of block size

all blocks in a single row, for example blocks in jth row with labels $b_{j,*}$ can be executed concurrently.

We select a block size for this partitioning such 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 a 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 to block for registers.

Processing of a block, for example, b_{32} requires m outputs each from blocks b_{22} and b_{23} that are processed as part of the previous iteration. A high-level description of the algorithm is shown in Algorithm 2. In our algorithm description, we use $neb(b_{j,i})$ to indicate the north-east boundary elements of $b_{j,i}$ and $nwb(b_{j,i})$ to indicate the north-west boundary elements of $b_{j,i}$. 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 $n + 1$. If these assumptions are not correct, the run times are changed by small constant factors.

Algorithm 2 $G^{(n)}_{biop}$

1: **for** $j = 2$ to $\frac{n+1}{m}$ **do**
2: {OpenMP thread directive is placed here}
3: **for** $i = 1$ to $\frac{n+1}{m} - j + 1$ **do**
4: processSquare$(b_{j,i}, neb(b_{j-1,i}), nwb(b_{j-1,i+1}))$
5: **end for**
6: **end for**

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 ith element of the shared array after processing nodes at level j holds the option price for node (j, i). The output of the neigbor $b_{j-1,i+1}$, north-west boundary, that is required for processing of $b_{j,i}$ is stored separately from the shared array. 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 $b_{j,i}$.

Observe that a large block size for L1 results in an unbalanced load distribution amongst cores. For a given problem size, there is an optimal block size for L1 as seen from the plots of Figure 10. The other factor that influences the load distribution is how various blocks are mapped to different cores. We use OpenMP directives to parallelize the computation across different cores. We use a 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 2).

Observe that processing of a block $b_{j,i}$ in line 3 requires input from blocks that were processed in previous iteration $j - 1$. Hence the blocks in the current iteration can be concurrently executed. The work load for each thread (core) is decided by the schedule directive of OpenMP.

There are three main types of schedules supported in OpenMP, namely static, dynamic, and guided. We experimented with different schedules and found that the static schedule with chunk size of one gave optimal performance. The chunk size greater than one for all schedules results in low performance due to unbalanced load distribution among cores. We use the *KMP_AFFINITY* environment variable to control binding of a thread to a physical core for optimal scheduling.

Analysis

We first estimate the memory traffic between an L2 cache and one of its children L1 caches. We 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 $nb = (n+1)/m$. 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.

Because there are eight cores, the number of blocks allocated to a core is approximately $n^2/16m^2$. A typical block has m^2 entries, m at the midpoint and $m(m-1)/2$ above and below the midpoint. To compute it requires that the northeast and northwest boundaries of two neighboring blocks be provided. The number of data items is $2m$. We assume that these values are not available in the L1 cache. The estimate for memory traffic between one L2 cache and one of its child L1

caches is given by

$$T_2^{(4)}/2 \approx 2m\left(\frac{n^2}{16m^2}\right)$$

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

From Theorem 7.2, the lower bound for $T_2^{(4)}$ is given by

$$T_2^{(4)} \geq \frac{\alpha_1 n(n+1)}{4\alpha_2^2\beta_1}$$

For our system $\alpha_1 = 8$, $\alpha_2 = 4$ and for $\beta_1 = 2m$ we get the following lower bound.

$$T_2^{(4)} \geq \frac{n(n+1)}{16m}$$

Thus the proposed algorithm performance can be 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 ignores load balancing issues. It may be possible to strengthen our bounds by considering these issues. Because our system has a large L2 cache for the problem sizes considered, 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 2 to 5. Our algorithm performs better for large problem sizes. We achieve 33% of the peak performance for 64K problem size versus 23% of the peak performance for the 8K problem size on 8 cores. For the 64K size problem, we obtained 16.7 GFLOPS. Similarly we observed a better scalability for large problem sizes. For example, we obtained a speedup of 7.3 for the 64K size problem on 8 cores versus a speedup of 5.5 for the 8K size problem on 8 cores.

For comparison, we also implemented a vanilla algorithm, which refers to a straightforward implementation of binomial 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 6. The best performance for the vanila algorithm is 4% of the peak as compared to 33% of the peak for our algorithm.

Table 2: Execution time for the binomial algorithm using OpenMP

	Execution Time (Seconds)			
n+1	1 Core	2 Cores	4 Cores	8 Cores
8192	0.094	0.050	0.028	0.017
16384	0.372	0.192	0.102	0.058
32768	1.414	0.723	0.379	0.208
65536	5.634	2.848	1.468	0.770

Table 3: Performance of the binomial algorithm as percentage of theoretical peak

	% Peak			
n+1	1 Core	2 Cores	4 Cores	8 Cores
8192	33%	32%	28%	23%
16384	34%	33%	31%	27%
32768	36%	35%	33%	30%
65536	36%	35%	34%	33%

Table 4: Performance of the binomial algorithm in GFLOPS

	GFLOPS			
n+1	1 Core	2 Cores	4 Cores	8 Cores
8192	2.1	4.0	7.2	11.7
16384	2.2	4.2	7.9	14.0
32768	2.3	4.5	8.5	15.5
65536	2.3	4.5	8.8	16.7

Table 5: Scalability performance of the binomial algorithm

	Speedup			
n+1	1 Core	2 Cores	4 Cores	8 Cores
8192	1.0	1.9	3.4	5.5
16384	1.0	1.9	3.6	6.5
32768	1.0	2.0	3.7	6.8
65536	1.0	2.0	3.8	7.3

Table 6: Performance of the vanilla-binomial algorithm using "-fast" and "-Qparallel" compiler options for optimization and auto-parallelization

n+1	Execution Time (sec)	GFLOPS	% Peak
8192	0.09	2.15	4.2%
16384	0.38	2.15	4.2%
32768	1.53	2.10	4.1%
65536	6.14	2.10	4.1%

9. CONCLUSIONS

In this paper, we present a unified memory hierarchy model for multicore architectures that have a varying degree of sharing of caches at different levels. We also present a general strategy that can help in deriving lower bounds for communication traffic for a single core and multiple cores for different applications. We show that the S-span of a DAG captures the computation dependencies inherent in the DAG and use it to develop lower bounds on communication traffic for a single core and multiple cores. We demonstrate that our unified framework can be applied to a variety of multicore architectures for a variety of applications.

We also give multicore algorithms for a financial application that exhibit a constant-factor optimal amount of memory traffic between different levels. We implemented these algorithms on a multicore system and demonstrated that our algorithms outperform compiler-optimized and auto-parallelized code by a factor of up to 7.3.

One of the limitations of our model is that it works for straight-line computations all of which can be represented as DAGs.

10. ACKNOWLEDGMENTS

This work was supported in part by NSF Grant CCF-0403674.

11. REFERENCES

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

[2] Teraflops research chip, 2008. `http://techresearch.intel.com/articles/Tera-Scale/1449.htm`.

[3] TILE64 processor family, 2008. `http://www.tilera.com/products/processors.php`.

[4] UltraSPARC T2 Processor – Overview, 2008. `http://www.sun.com/processors/UltraSPARC-T2/`.

[5] 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):563–576, 1994.

[6] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of 19th Annual ACM Symposium on the Theory of Computing*, pages 305–314, New York, 1987.

[7] A. Aggarwal, A. Chandra, and M. Snir. Hierarchical memory with block transfer. In *28th Annual Symposium on Foundations of Computer Science*, pages 204–216, Los Angeles, California, October 1987.

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

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

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

[11] 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*, pages 116–123, 1993.

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

[13] J. C. Cox, S. A. Ross, and M. Rubinstein. Option pricing: A simplified approach. *Journal of Financial Economics*, 7(3):229–263, September 1979.

[14] 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 Not.*, 28(7):1–12, 1993.

[15] J. J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson. An extended set of Fortran basic linear algebra subprograms. *ACM Trans. on Math. Soft.*, 14:1–17, 1988.

[16] 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 Trans. Math. Soft.*, 14:18–32, 1988.

[17] S. Fortune and J. Wyllie. Parallelism in random access machines. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118, New York, NY, USA, 1978. ACM.

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

[19] 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):10–24, 2006.

[20] 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 Trans. Math. Softw.*, 24(1):74–101, 1998.

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

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

[23] J.-W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proc. 13th Ann. ACM Symp. on Theory of Computing*, pages 326–333, 1981.

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

[25] 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*, pages 240–251, London, UK, 1994. Springer-Verlag.

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

[27] Y. Kwok. *Mathematical Models of Financial*

Derivatives. Springer-Verlag, Singapore, 1998.

[28] 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 (HICSS'95)*, page 51, Washington, DC, USA, 1995. IEEE Computer Society.

[29] J. E. Savage. Extending the Hong-Kung model to memory hierarchies. In D.-Z. Du and M. Li, editors, *Computing and Combinatorics*, pages 270–281. Springer-Verlag, Lecture Notes in Computer Science, 1995.

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

[31] J. E. Savage and M. Zubair. Cache-optimal algorithms for option pricing, 2008. Submitted for publication.

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

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

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

[35] 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*, pages 159–169, New York, NY, USA, 1990. ACM.

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

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