An Early Evaluation of the Scalability of Graph Algorithms on the Intel MIC Architecture

Erik Saule* and Ümit V. Çatalyürek*†
* *Department of Biomedical Informatics*
†*Department of Electrical and Computer Engineering*
The Ohio State University
Email: {esaule,umit}@bmi.osu.edu

***Abstract*—Graph algorithms are notorious for not getting good speedup on parallel architectures. These algorithms tend to suffer from irregular dependencies and a high synchronization cost that prevent an efficient execution on distributed memory machines. Hence such algorithms are mostly parallelized on shared memory machines. However, current commodity shared memory machines do not typically offer enough parallelism to process these problems. In this paper, we are presenting an early investigation of the scalability of such algorithms on Intel's upcoming Many Integrated Core (Intel MIC) architecture which, when it will be released in 2012, is expected to provide more than 50 physical cores with SMT capability. The Intel MIC architecture can be programmed through many programming models, here we investigate the three most popular of these models namely OpenMP, Cilk Plus and Intel's TBB. We present scalability results of a parallel graph coloring algorithm, three variations of a breadth-first search algorithm and a microbenchmark for irregular computations using these three programming models. Our results on a prototype board show that the multi-threaded architecture of Intel MIC can be effectively used for hiding latencies in irregular applications to achieve almost perfect speedup.**

***Keywords*-Graph algorithm; unstructured irregular computation; scalability; multi-threaded architectures; graph coloring; breadth-first search**

I. Introduction

The Intel Many Integrated Core (Intel MIC) architecture is Intel's latest design targeted for processing highly parallel workloads. Built on simple x86 cores, the design allows to use standard, existing programming tools and methods. The current prototype Intel MIC cards, codenamed Knights Ferry (KNF), provides in a single chip up to 32 cores with 4-way SMT, where each core features 512 bit-wide SIMD registers. The chip also includes coherent L1 and L2 caches and the inter-processor network is a bidirectional ring. The final commercial design, codenamed Knights Corner, will feature more than 50 cores which, undoubtedly, will make the Intel MIC architecture a very attractive component for traditional high performance computing (HPC) environments. The computational kernels of many traditional HPC applications have usually *regular* dependencies, and can be *vectorized* very easily, which will be an ideal fit for the Intel MIC architecture. One can expect that in the following years we will see many study dedicated to the use the Intel MIC architecture for such regular numerical computation.

However, in this work, our focus will be irregular applications, in particular we will investigate the scalability of shared-memory graph algorithms on the Intel MIC architecture. Graph algorithms are notorious for being hard to efficiently parallelize on due to their irregular dependencies [1]. Here, we investigate the scalability of two important graph algorithms, namely graph coloring and breadth-first search, and a microbenchmark for further investigating the interplay of SMT, arithmetic unit and memory system on a prototype of the Intel MIC architecture.

Graph coloring is a combinatorial problem which consists in partitioning a graph in a minimum number of independent sets. This problem has numerous applications, the most known use in parallel computing is to represent the tasks of a computation as the vertices of a graph, and an edge connects two vertices if these two vertices cannot be computed simultaneously [2]. Finding a coloring of this graph allows to partition the tasks into sets that can be safely computed in parallel. Minimizing the number of colors decreases the number of synchronization points in the computation and increases the efficiency of the parallel platform. Other applications of graph coloring appear in automatic differentiation [3] and parallel numerical computation [4] A variant of coloring called distance-2 coloring has many applications including some in the compression of Jacobian and Hessian matrices for sparse linear algebra [3].

Breadth-first search (BFS) is an archetypical graph traversal technique which enumerates the vertices of the graph in the order of their "distance" from a *source* vertex. BFS implicitly computes shortest paths from a source vertex and is a generic kernel many algorithms are based on [5], including computationally expensive centrality measures [6]. Computing BFS in parallel is known to be a difficult challenge, which is why it is one of the reference graph algorithm of the Graph 500 benchmark[1].

Both coloring and BFS kernels can be classified as memory-intesive, since there is little to no computation

[1] http://www.graph500.org/

in these kernels. Not all irregular applications are free of floating-point computations. For example, in simulations that use unstructured mesh computations [7], dependencies on neighboring mesh elements make the structure of computations irregular. That is, similar to graph kernels, in these applications visiting neighbor elements are required and such visits involve some additional floating-point computations. Hence in such irregular applications, the computation to communication ratio could be significantly different. We have implemented a simple microbenchmark to investigate the effects of the computation to communication ratio in irregular applications.

These three kernels cover a wide range of irregular applications. Therefore we believe they are representative of the performance we could achieve on any irregular application using an Intel MIC coprocessor. In this paper, we investigate for each of these kernels at least three implementations using different algorithms and using different programming models, namely OpenMP, Cilk Plus and Intel's TBB. Our goals in this work are to

- evaluate development efforts of graph algorithms in three different programming models: OpenMP, Cilk Plus and TBB,
- investigate the ease of porting applications to Intel MIC,
- evaluate the performance of the runtime engines that support the three programming models we investigate,
- evaluate the scalability performance of graph algorithms on the Intel MIC architecture,
- investigate algorithm engineering required to optimize algorithms to scale beyond the parallelism provided by current CPU architectures.

Our findings are that there is little effort required to port a code from regular CPUs to Intel MIC and that the obtained scalability of an algorithm on Intel MIC is derived from the scalability of the application on CPUs. We report linear speedup up to the maximum number of threads on graph coloring when the memory subsystem is stressed, thanks to the overlapping of memory latency by the SMT. For high computation to communication ratio, the irregular computation kernel displays linear speedups up to the number of cores, and the speedup up continues to increase when more that one thread per core is used, indicating the overlapping of floating point computation and memory transfers by the SMT.

On BFS, our investigation also led us to develop a more realistic performance model for the BFS algorithm, as well as a novel reengineering of the algorithm that uses block-accessed queues. We report, although sub-linear, a speedup that matches the proposed performance model. In other words, the execution of our new algorithm on Intel MIC uses all the parallelism the algorithm can offer.

II. Programming Frameworks

A. OpenMP

OpenMP[2] is an API that supports multi-platform shared memory parallel programming in C, C++ and Fortran. It consists of a set of compiler directives and library routines, where runtime behavior can be controlled by environment variables. Compiler directives are used to annotate loop bodies and sections of the codes for parallel execution and marking variables as local, or shared (global). Some constructs exist for critical sections, declaring completely independent tasks, or doing reductions on variables.

When a *for* loop is declared to be parallel, the iterations of the loop are executed concurrently. The iterations of the loop can be allocated to the working threads according to three scheduling policies: *static*, *dynamic*, and *guided*. In static scheduling, the iterations are either partitioned in as many intervals as threads or partitioned in *chunks* which are allocated to the threads in a round-robin fashion. The dynamic scheduling partitions the iterations in chunks, where chunks are dynamically allocated to the threads using a First-Come First-Serve policy. Finally, the guided scheduling policy tries to reduce the scheduling overhead by allocating first a large amount of work to each thread and geometrically decreases the amount of work allocated to the thread (up to a given minimum chunk size) in order to optimize the load balance.

B. Cilk Plus

Cilk Plus[3] is an Intel derivative of the Cilk C extension from MIT [8]. The core programming model of Cilk Plus follows the asynchronous function call semantic. Basically, function calls can be tagged with the Cilk_spawn keyword that indicates that the function can be executed concurrently to the current function. A function can wait for the completion of all the functions it spawned using the Cilk_sync keyword. Cilk allows to easily leverage nested parallelism, which is only a recent addition to the OpenMP standard and often reported to perform poorly.

The tasks are executed by the runtime system within a work-stealing framework. Cilk uses a double ended queue per thread to keep track of the tasks to execute and uses it as a stack during regular operations conserving a sequential semantic. When a thread runs out of task, it *steals* the deepest half of the stack of another (randomly selected) thread. This scheduling policy has been shown multiple times to provide close to optimal load balance [8], [9], [10].

Cilk Plus provides more constructs than the original Cilk. It allows for instance to easily execute a for-loop in parallel using a recursive task decomposition, through the Cilk_for construct. It also provides data parallel operations on regular

[2] http://openmp.org/wp/
[3] http://software.intel.com/en-us/articles/intel-cilk-plus/

C or C++ arrays, either by the use of *array* notation[4], intrinsic functions or user defined *elemental* functions.

All the variables declared inside a scope are local to that thread, while all the other ones might be shared with other thread. Thread Local Storage and reductions are performed through *holders* and *reducers* [11]. A user can define her own Thread Local Variable by implementing a *monoid* which allows to define what should happen during a *steal* and a *reduce* operations.

C. Threading Building Blocks

Intel Threading Building Blocks (TBB) is a versatile parallel programming framework which contains tools such as scalable memory allocators, cache aligned arrays, abstraction for atomic operations, thread safe containers, and so on. But TBB is mainly useful for its parallel constructs which are typically executed using a work-stealing framework.

The *flow graph* construct allows to define tasks that are repeatedly executed by taking some data as an input and producing an output. It allows to easily set up a pipeline of tasks that perform complex tasks such as, typically, video compression, graphical rendering, and data processing.

The other constructs take *splitable* objects as parameter which represents some parts of the work which can computed and eventually partitioned in to two. In particular, the parallel_for construct takes as parameter a *range* and a partitioning policy. The policy is applied to cut the range in small parts which are executed. Notice that the partitioner is inside the scheduling loop and can then take runtime decision.

Three partitioners are natively available in TBB. The *simple* partitioner recursively divides the *range* until a given size is reached. In a way, the *simple* partitioner is similar to the *dynamic* scheduling policy of OpenMP. The *auto* partitioner uses the work stealing events to decide whether to split a *range* or not. Basically, it creates some subranges first and subdivide a range further only when it gets stolen. Finally, the *affinity* partitioner tries to allocate iterations of a range to maximize cache usage by trying to allocate consecutive iterations to the same thread. If the same *affinity* partitioner is used on multiple loops, it tries to allocate the iterations to the thread that executed them during the previous loop.

III. Algorithms

A. Graph Coloring

The distance-1 coloring problem is formally defined as follows. Let $G = (V, E)$ be a graph with $|V|$ vertices and $|E|$ edges. The set of neighbors of a vertex v is $adj(v)$; its cardinality, also called the *degree* of v, is δ_v. The degree of the vertex having the most neighbor is $\Delta = \max_v \delta_v$. A coloring $C : V \rightarrow \mathbb{N}$ is a function that maps each vertex of the graph to a color (represented by an integer), such that two adjacent vertices have different colors, i.e., $\forall (u, v) \in E, C(u) \neq C(v)$. Without loss of generality, the number of colors used is $\max_{u \in V} C(u)$. The optimization problem at hand is to find a coloring with as few colors as possible and is NP-Hard for arbitrary graphs [12]. Recently, it has been shown that, for all $\epsilon > 0$, it is NP-Hard to *approximate* the graph coloring problem within $|V|^{1-\epsilon}$ [13].

Despite the pessimistic theoretical results and the existence of more complicated algorithms, for many graphs that arise in practice, solutions that are provably optimal or near optimal can be obtained using a simple *greedy* algorithm [14]. In this algorithm, the vertices of the graph are visited in some *order* and the smallest permissible color at each iteration is assigned to the vertex. The pseudocode of this technique is presented in Algorithm 1. Choosing the smallest permissible color is known as the *First Fit* strategy. This simple algorithm has two nice properties. First, for any ordering of the vertices, it produces a coloring with at most $1 + \Delta$ colors. Second, for some orderings of the vertices it will produce an optimal coloring [15].

Algorithm 1: SeqGreedyColoring

```
Data: G = (V, E)
maxcolor ← 1
for each v ∈ V do
    for each w ∈ adj(v) do
        forbiddenColors[color[w]] ← v
    color[v] ← min{i > 0 : forbiddenColors[i] ≠ v}
    if color[v] > maxcolor then
        maxcolor ← color[v]
return maxcolor
```

Although the simple greedy algorithm is very effective, it is also very sequential in nature. Gebremedhin and Manne [16] proposed *speculation* as an alternative strategy for coping with this problem. The main idea is to speculatively color as many vertices as possible concurrently, tentatively tolerating potential conflicts, and detect and resolve conflicts afterwards. In their basic shared-memory algorithm while the coloring and the conflict detection are done in parallel, the conflict resolution is done sequentially. Bozdağ et al. [2] extended the algorithm in [16] to make it suitable for and well-performing on distributed memory architectures. One of the extensions was replacing the sequential recoloring phase with a parallel iterative procedure. Later, Çatalyürek et al. [17] presented a careful implementation of the coloring algorithm on various multi-threaded architectures, that include a 128-processor Cray XMT, a 16-core Sun Niagara 2, and an 8-core Intel Nehalem system. In this paper, we use the OpenMP implementation of Iterative Parallel Greedy Coloring algorithm from that work [17]. We also developed Cilk Plus and TBB implementations of

[4]such as *w[:] = a*x[:]+b*y[:]*

the same algorithm for comparison of the runtime systems of these three programming models. The pseudocodes of the algorithms are given in Algorithms 2–4. The graph is traversed at least twice: once for coloring and once for detecting eventual conflicts, hence the potential gain by using a second execution unit may not be fully realized due to this additional conflict resolution phase.

Algorithm 2: PARITERATIVECOLORING

```
Data: G = (V, E)
Visit ← V
color[v] ← 0 for v ∈ V
while Visit ≠ ∅ do
    maxcolor ← PARTENTATIVECOLORING(G, Visit, color)
    Visit ← PARDETECTCONFLICT(G, Visit, color)
return maxcolor, color
```

Algorithm 3: PARTENTATIVECOLORING

```
Data: G = (V, E), Visit ⊂ V, color[1 : |V|]
maxcolor ← 1
localMC ← 1
for each v ∈ Visit in parallel do
    for each w ∈ adj(v) do
        localFC[color[w]] ← v
    color[v] ← min{i > 0 : localFC[i] ≠ v}
    if color[v] > localMC then
        localMC ← color[v]
maxcolor ← Reduce(max) localMC
return maxcolor
```

Algorithm 4: PARDETECTCONFLICT

```
Data: G = (V, E), Visit ⊂ V, color[1 : |V|]
Conflict ← ∅
for each v ∈ Visit in parallel do
    for each w ∈ adj(v) do
        if color[v] = color[w] then
            if v < w then
                atomic Conflict ← Conflict ∪{v}
return Conflict
```

B. Irregular Computation Microbenchmark

Our irregular computation microbenchmark is based on a simple traversal of the computational dependency graph. In the computational dependency graph, each vertex represents an atomic task (such as mesh element update) and its adjacency represents the dependencies to its "neighbor" elements. In our microbenchmark, each vertex of the graph has some state. During the computation, the state of each vertex is updated based on the state of its neighbors. Although in real life applications computations might involve complex functions, since we are only interested to investigate effects of the interplay between memory system and CPU, we will use a simple loop over neighbors with varying intensity, to parametrize the computation to communication ratio. That is, we implemented a simple microbenchmark kernel where the state of a vertex is a double precision floating point and we simply average the state of the neighbors. In order to see the effect of the amount of computation carried, we will perform this iteration multiple ($iter$) times. Algorithm 5 shows the pseudocode of our microbenchmark kernel. Notice that this algorithm is a reasonable abstraction of a single iteration of algorithms such as Page Rank or Heat Equation solvers and has data dependencies similar to a sparse matrix vector multiplication.

Algorithm 5: IRREGULARCOMPUTATION

```
Data: G = (V, E), Visit ⊂ V, state[1 : |V|]
for each v ∈ V in parallel do
    for i = 0; i < iter; i++ do
        sum ← state[v]
        for each w ∈ adj(v) do
            sum ← sum + state[w]
        state[v] ← sum / (|adj(v)| + 1)
```

C. Parallel Breadth-First Search

The sequential BFS algorithm is a simple, yet archetypical graph search algorithm [5]. Many important graph kernels uses ideas similar to BFS. It consists in enumerating the vertices of a graph in their order of distance to a reference vertex, also known as *source* vertex. In a sequential setting, BFS is implemented by setting the level of the reference vertex to one and adding the vertex to a First In First Out (FIFO) queue. Then the algorithm iterates over the queue, and for each extracted vertex, it enumerates the neighbors and, if they have not been seen yet, set their level and add them to the queue. Algorithm 6 shows the pseudocode of sequential BFS.

Algorithm 6: SEQBREADTHFIRSTSEARCH

```
Data: G = (V, E), source ∈ V
for v ∈ V do
    bfs[v] ← −1
bfs[source] ← 0
FIFO.push(source)
while ! FIFO.empty() do
    v ← FIFO.pop()
    for each w ∈ adj(v) do
        if bfs[w] = −1 then
            bfs[w] ← bfs[v] + 1
            FIFO.push(w)
return bfs
```

One of the possible ways to parallelize BFS is to iterate over the levels in multiple synchronous steps. This gives us a layered parallel BFS algorithm which have been exploited in both distributed memory [18] and in shared memory [19], [20]. Algorithm 7 presents a simple generic pseudocode. At each step, all the vertices that belong to one level are considered. The level of all the neighbors of the considered vertices that have never been seen before are set to the next level and are added to the set of vertices to be considered at the next level. Notice that the layered structured of the BFS allows to determine the level of w without having to read the level of v avoiding some memory accesses.

Algorithm 7: PARLAYEREDBFS

```
Data: G = (V, E), source ∈ V
for v ∈ V in parallel do
    bfs[v] ← −1
bfs[source] ← 0
cur.add(source)
level ← 1
while ! cur.empty() do
    for v ∈ cur in parallel do
        for each w ∈ adj(v) in parallel do
            if bfs[w] = −1 then
                bfs[w] ← level
                uniquely next.add(w)
    SWAP (cur, next)
    level ← level + 1
return bfs
```

Ideally, one want the addition of a vertex to the next set to be done so that a given vertex appears only once in the set to avoid traversing that vertex multiple time. There are mainly two possible ways of ensuring that property, either the access to bfs are done atomically, or the data structure of next avoid duplicates. Both operations are typically expensive. Leiserson and Schardl [20] noticed that the race condition is unlikely and benign. Indeed, two threads can modify the bfs array concurrently, but whichever wins the race leads to the same values in memory. Two threads adding the same vertex to next is also benign. Redundant computations will be carried at the next level, but the algorithm stays valid. Moreover, the excessive computations most likely do not snowball. That is, in a shared memory setting, even if a vertex is visited by two (or more) threads in the next level of BFS, it is very unlikely that their neighbors will be added to queue multiple times, which could only happen when there is a race condition when their levels are checked before adding them to the queue. In our work, we have also observed experimentally that the redundant computation time is easily amortized by the savings from the elimination of additional synchronization (see Section V).

The performance of the BFS algorithm drastically depends on the structure of the graph. In an extreme case, consider a graph that is a very long chain, the layered BFS algorithm will not be able expose any parallelism if we start searching from one end of the graph. There are two sources of parallelism in this algorithm, for a given level, the vertices can be traversed in parallel, and for a given vertex, its neighbors can be traversed in parallel. In many scalable implementations (and all the ones we consider), the parallel execution is organized in blocks of vertices within a given level, since a finer parallelism will induce a high scheduling overhead.

To understand these two phenomena, we propose a simple model of the layered BFS algorithm. The computations are decomposed in L synchronized parallel steps, one step for each level of the BFS. There are x_l vertices to visit in the lth level of the BFS. We assume the computation is performed by t threads using blocks of vertices of size b. We make the following five (unrealistic) assumptions: each vertex is processed in the same time, there are no cache effects, processing threads are completely independent, there is no scheduling or synchronization overhead. The computation time of level l is then: $c(l) = x_l$ if $x_l < b$ and $c(l) = \lceil \frac{x_l}{tb} \rceil * b$ otherwise. This reflects that if there is less than one block of work a single thread will execute it. If there is more than one block, multiple threads will execute them in $\lceil \frac{x_l}{tb} \rceil$ rounds, each of them taking b time units. For a given block size, the achievable speedup is computed as $\frac{\sum_{l=1}^{L} x_l}{\sum_{l=1}^{L} c(l)}$.

IV. IMPLEMENTATION

Programming for Intel MIC is done similarly to programming for a regular CPU. Actually all the codes we developed can either be compiled for Intel MIC or for a regular x86 CPU architecture. The only difference is one line of code annotation that expresses that a given block should be executed on the Intel MIC coprocessor, with the list of variables that should be copied onto the memory of the coprocessor. Below we will describe the implementation details of our graph algorithms using the three programming models we have described in Section II.

A. Graph Coloring

For the graph coloring problem, we will examine three different implementations of the parallel iterative graph coloring algorithm given in Algorithm 2–4, using the selected three programming models. We would like to note that, we do not claim that this is the best algorithm for all three programming models, and indeed, our algorithm had been originally developed for OpenMP type of parallelism. Implementations using different programing models will enable us to investigate the performance of the runtime systems supporting these programing models.

In our algorithm there are four central issues that need to be addressed: execution of the two *for* loops in parallel, accessing the localFC, reducing maxcolor and properly

implementing the Visit/Conflict data structure. For the last one, we simply use two arrays of size $|V|$ and since the number of conflicting vertices is usually low, we use an atomic fetch_and_add to obtain a unique index in the Conflict array. We will discuss how we handle the other three in each programming model below.

1) OpenMP: Both *for* loops are executed using the *parallel for* construct of OpenMP. The *parallel for* can be executed with different scheduling policies and chunk size (see Section II-A), leaving multiple variant to test for.

In the tentative coloring, the localFC are stored contiguously in memory (but without sharing a cache line). At the beginning of the parallel section, each thread obtains a pointer on a different localFC using their thread IDs as an offset.

At the time the code was written, the OpenMP 3.0 standard in C and C++ does not provide a *max* reducing operation as it does in Fortran[5]. So the computation of the number of colors is performed manually by having each thread maintain a local maximum (using the same indexing technique used for localFC). The values are reduced at the end of the parallel section by the main thread.

2) Cilk Plus: The Cilk Plus version of the coloring code uses the Cilk_for construct to create parallelism. Given an interval, the construct recursively spawns two tasks that process half the iterations of the loop until a given chunk size is obtained. By default Cilk Plus sets the chunk size so that the number of tasks is proportional to the number of threads. This rule leads to a good load balance in many cases. However, one can set the granularity size manually.

Obtaining a localFC array can be done in two different ways. The first one is a similar to the OpenMP implementation. It is possible, but discouraged, to obtain a unique worker ID using __cilkrts_get_worker_number()but that ID can be much larger than the number of threads involved in the computation. An upper bound on the worker IDs is exposed but using that number results in initializing more memory necessary.

The Cilk Plus way of obtaining a localFC array is to use a *view* which are the building block behind *reducers*. Basically, a *view* is a thread local variable that is initialized for a thread at the time it uses it. The local variables are reduced automatically during merge operations. A program can define it own initialization and reduction operation, allowing to allocate memory on demand. The drawback of this approach compared to the one using worker number is that memory is allocated and initialized during the processing of the computation, potentially increasing load imbalance. This variant will be referred to as the variant with a holder.

The computation of maxcolor is performed using a reducer_max object. Intuitively, reducer objects store a local state of a given variable and can only be accessed using a write only semantic. A local copy of the variable is created during steal operations and the copies are reduced during merge operations. The reduce operation should allow to obtain a final result identical to a sequential execution. However it is not always possible if the operations are not bit-wise commutative (such as floating point operations)

3) TBB: The TBB version of the algorithm uses the *parallel_for* construct. This construct is a function call that takes as parameter three objects. The first one defines the range of iterations to process and how they can be split; here we use a block_range that allows to set a chunk size under which TBB will not partition further. The second one is a functor (an object function) that will be applied on partitioned ranges which in our case is basically the loop body. The last one is the partitioning policy which is set to one of the scheduling policy described in Section II-C. This leads to three different variants of our TBB implementation.

TBB does not expose thread number to the developer. It is therefore impossible to access the localFC array as it is done in Cilk Plus and OpenMP using thread IDs. (Though there are some other workarounds to create thread-level storage which we exploited in the BFS code). However, TBB provides a construct similar to *views* in Cilk which are called enumerable_thread_specific. At most one object per thread is created on demand and we can obtain a localFC using a type that dynamically allocates the right amount of memory.

The TBB way of performing reduction (for maxColor) is to use the *combinable* construct. When a thread has a combinable object, it obtains its own copy of the variable. At the end of the parallel execution, one can call the *combine* function to perform an aggregation of the different values by passing a binary functor as a parameter.

B. Irregular Computation

The irregular computation is simple kernel with just a *for* loop that needs to be parallelized. We implemented this simple kernel using the same techniques used for the graph coloring algorithm.

C. Breadth-First Search

For BFS, to highlight the importance of algorithm and data structure for scalability, instead of implementing the same algorithm on different programing models, we used three variants of the BFS algorithm. At the high level, all use the parallel layered BFS algorithm described in Algorithm 7, hence the amounts of computations carried out by these algorithms are similar. But the three variants use different data structure for keeping track of the vertices to be visited in the next level.

The first one is the algorithm from Leiserson and Schardl [20]. It uses a smart bag data structure that allows to split and merge vertex lists fast. There are multiple theoretical guarantees. The MIT Cilk implementation of the

[5]The OpenMP 3.1 standard accepted in July 2011 now provides such a reduction.

algorithm is available online[6] and we ported the code to Intel Cilk Plus by simply substituting the header files of MIT Cilk with the ones of Cilk Plus. The bag data structure consists of arrays of balanced trees of size 2^k. For each k, the bag contains at most one tree of that size. Such an organization allows to easily merge two bags together by using an algorithm similar to carry-add for integer addition. The code utilizes dynamic memory for its bag data structure and uses complex pointer techniques to manage its memory. To avoid a high cost of operation, the node of the balanced tree can store more than a single element (this parameter is called *grainsize* in [20]). This algorithm does not strictly enforce that each vertex is uniquely added to its bag data structure. We will refer to this variation of the queue as *relaxed* queues. It is shown that the effects are benign and on the overall, even though the code may carry out some redundant computations, this scheme yields some performance improvements. We will refer to this bag-based Cilk code as CilkPlus-Bag-relaxed.

The second implementation we used is from SNAP v0.4[7] which uses thread-local storage (TLS) to concurrently keep partial queues in each thread to avoid synchronization overheads, and then at the end of each level, local queues are merged into a global queue. It is implemented using OpenMP. This algorithm locks a vertex before adding it to local queue to guarantee that only one instance of that vertex will be added to any local queues [19]. We carried one small improvement, by checking if a vertex is traversed before attempting to lock it. This code base will be called OpenMP-TLS.

We developed a novel block-accessed shared queue data structure for a scalable shared memory layered BFS algorithm. The block-accessed queue uses contiguous memory (simply an array) for storing vertices to be visited in the next level of BFS. To avoid locking too often, each thread reserves a block of memory from the queue and uses that block for adding vertices to the next level. When a thread's block is full, it grabs a new block. This operation can simply be implemented using an index pointer pointing to the next available block and a new block is obtained by incrementing the index pointer using an atomic fetch_and_add of the block size. At the end of a level, it is possible that some threads have not entirely filled the last block they were operating on, hence there will be un-initialized space in the queue. One approach is to compact the queue by swapping the last filled elements with these spaces, but this requires a complex book keeping data structure. Instead, we fill the remaining of the block with a sentinel value (an invalid vertex ID, such as -1) to indicate that it is not a vertex that needs to be visited. Hence in the vertex visit loop, the value needs to be checked if it contains a valid vertex ID. Although this scheme can produce slightly larger queues, by keeping the block size small (but not so small so that we do not use atomics too often), the overhead is minimized. We have implemented both OpenMP and TBB BFS codes that uses block-accessed queue, which we will call OpenMP-Block and TBB-Block, respectively. We have also used trick presented in Leiserson and Schardl [20] to relax the vertex addition to queue, and these variants are called OpenMP-Block-relaxed and TBB-Block-relaxed.

[6] http://web.mit.edu/~neboat/www/code.html

[7] http://snap-graph.sourceforge.net/

V. Experiments

A. Setup

The experiments are run on seven real-world application graphs that come from various application areas including linear car analysis, finite element, structural engineering and automotive industry [16], [21]. They have been obtained from the University of Florida Sparse Matrix Collection[8] and the Parasol project. The list of the graphs and their main properties are summarized in Table I. The number of colors obtained with a sequential run of the greedy algorithm is also listed in the table, as well as the number of levels of a BFS traversal from vertex number $\frac{|V|}{2}$.

Name	$\|V\|$	$\|E\|$	Δ	#Color	#Level
auto	448K	3.3M	37	13	58
bmw3_2	227K	5.5M	335	48	86
hood	220K	4.8M	76	40	116
inline_1	503K	18.1M	842	51	183
ldoor	952K	20.7M	76	42	169
msdoor	415K	9.3M	76	42	99
pwtk	217K	5.6M	179	48	267

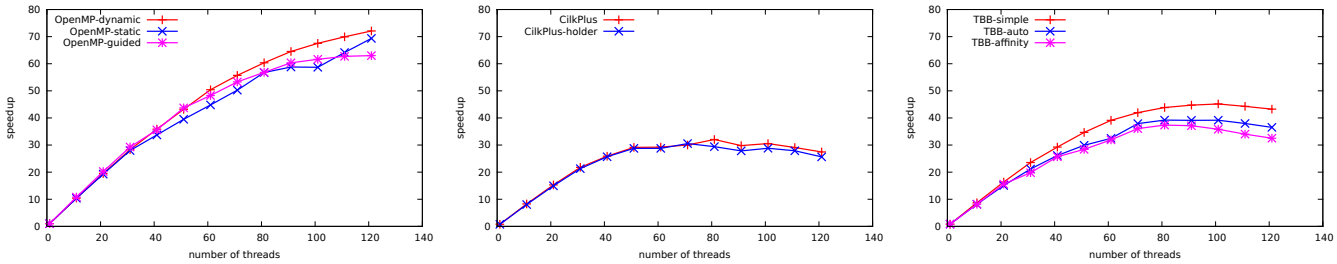
Table I
Properties of the test graphs.

The prototype KNF Intel MIC card we use exposes 31 computational cores when used in offloaded mode (32 are on the chip but one is reserved by the system), each featuring 4-way SMT. There is 1GB of GDDR5 memory available. The operating system running on the card is a derivative of FreeBSD. All the codes that run on KNF are compiled using a version of icc for Intel MIC. The host uses a dual Intel Xeon X5680 CPUs clocked at 3.3Ghz with 24GB of main memory and runs CentOS 5.5. Codes compiled on the host are typically compiled with icc. We also performed test compiling with gcc 4.5.6 for all the codes that uses OpenMP and TBB and report the best result[9]. The only case where gcc 4.5.6 was leading to better performance is when compiling SNAP which are reported as OpenMP-TLS in Figure 4(d).

We would like to note that no absolute numbers will be quoted for two reasons. First, the focus here is on scalability and absolute numbers are likely to be meaningless since

[8] http://www.cise.ufl.edu/research/sparse/matrices/

[9] Cilk Plus has recently been added to more recent versions of GCC



(a) Using OpenMP (b) Using Cilk Plus (c) Using TBB

Figure 1. Speedup of the coloring implementations on all (naturally ordered) graphs.

KNF is a prototype design. Second, at the time of writing this manuscript, the numbers are confidential and the authors are under non-disclosure agreement.

In all experiments, 10 runs are performed we report the average of the performance or the last 5 runs. This allows to be sure that the runtime systems are properly initialized. The graph is transfered to the memory of the co-processor beforehand. Speedup value on multiple graphs are geometric mean of the speedup of each graph, which is computed using as baseline the configuration that performs the fastest on 1 thread for that graph.

B. Graph Coloring

We start the evaluation of the coloring algorithms by testing independently the different variants and parameters of each implementation (see Figure 1). The evaluation is performed using a number of threads from 1 to 121 by increment of 10. Different chunk sizes (from 40 to 150) were tried and only the best results are reported.

We observed that, for the OpenMP experiments, the dynamic scheduling policy performs better with a chunk size of 100. The static policy is better with a chunk size of 40 and the guided scheduling policy performs better with a chunk size of 100. Figure 1(a) presents the results of the best configuration for each scheduling policy. The three scheduling policies lead to similar results on 31 threads and less. After 51 threads, the dynamic scheduling clearly appears to be better than the guided and static scheduling policies. The guided scheduling policies appears to be in general, but not always, better than the static scheduling policy. Therefore, the dynamic variant will be the one reported for coloring with OpenMP.

We have two variants of the Cilk Plus implementation of the coloring algorithm (see Section IV-A2): one uses a holder and the other one uses thread id to access the local storage. The dynamic nature of the Cilk Plus scheduler makes it difficult to clearly state which parameter set is better. Figure 1(b) presents the performance achieved by both variants run with a granularity of 100 which was the chunk size that leads to the highest speedup. The performance of both variants are very close. Since the use of worker IDs is discouraged, the variant with the holder will be the one reported for the Cilk Plus implementation of coloring.

There are three implementations of the coloring algorithm using the TBB programming model. They differ in the partitioner used. All three variants reached best performance when run with a minimum chunk size of 40. Figure 1(c) presents the results obtained using the three partitioners. The simple partitioner clearly leads to better speedup in this experiments on 31 threads and more. The affinity partitioner seems to perform consistently slower than the auto partitioner. From now on, the result with the simple partitioner will be the one reported for the TBB implementation of the coloring algorithm.

As seen in Figure 1 our OpenMP implementation clearly scales better than our TBB and Cilk Plus implementations. The speedup achieved by the OpenMP implementation monotonically increases with the number of threads. It achieves a speedup of 60 using 81 threads and reach a speedup of 72 on 121 threads. The speedup achieved by the TBB implementation peaks at a speedup of 45.15 achieved on 101 threads. Our Cilk implementation peaks at a speedup of 32 obtained when using 81 threads.

Parallelization of the coloring algorithm would not be useful if the quality of the solution was significantly degraded. We verified that the number of colors never differ by more than 5% when the algorithm is executed in parallel.

We have observed that our codes achieved speedup up to 121 threads despite there is only 31 cores on our prototype Intel MIC coprocessor. The SMT are providing a significant improvement, indicating that the memory subsystem is stressed. To significantly increase the amount of memory transfers (and the runtime), we shuffled the vertex IDs of graphs randomly which break all the locality that naturally appears in the graphs. The same variants of the coloring algorithm proved to be best and we only report the best speedup achieved by each variant in Figure 2. The OpenMP implementation reaches a speedup of 153 despite there are only 121 threads used. The TBB and Cilk Plus implementations reach a speedup of 121 and 98, respectively.

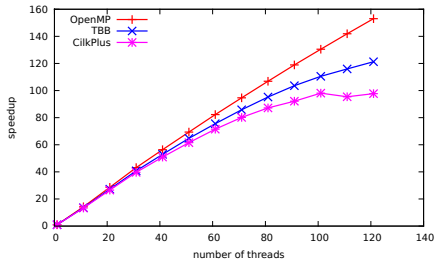


Figure 2. Speedup of coloring on the randomly ordered graphs.

The graph coloring kernel stresses the memory subsystem. A proper use of the SMT capabilities of the Intel MIC architecture is the key to hide latencies and achieve best performance. Therefore, it comes without a surprise that the less expensive dynamic scheduling policies performs better than the more complex ones. On randomly ordered graphs, the obtained speedup are linear in the number of threads showing that the memory subsystem of Intel MIC scales well.

C. Irregular Computation

This experiment is designed to vary the computation requirement of the application so as to test the performance of Intel MIC on irregular applications that are less memory intensive than graph coloring. The results of our experiments are presented in Figure 3. Since the number of iterations varies, hence the amount of computation, the speedup are computed relatively to the same number of iterations. In this experiment OpenMP performed best using the dynamic scheduling policy and TBB performed best using the simple partitioner.

When increasing the amount of computation the speedup of the OpenMP and TBB implementations decrease (See Figure 3(a) and 3(c)). Indeed, when we increase the amount of computation we also increase the contention on the FPU, so the SMT becomes less important and speedup decrease. Interestingly, the speedup of Cilk Plus (see Figure 3(b)) increases with the computation since an increase in the amount of computation reduces the scheduling overhead.

Eventually, with 10 iterations the three programing models (and runtime system) reaches essentially the same performance. The highest speedup is 49 and is obtained using 121 threads. The speedup only marginally increased compared to the 61 threads case where the speedup was 46.

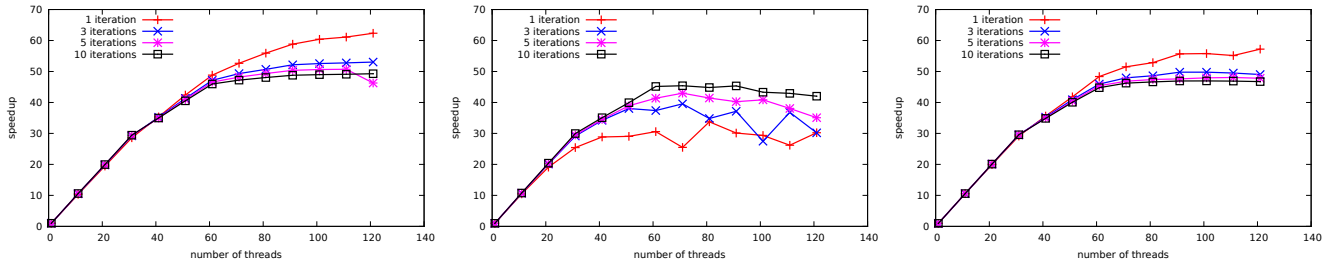
When the amount of computation increases the speedup achieved decreases since the pressure on computational components within the cores are more stressed and SMT becomes less useful. Yet, SMT can not be ignored since the speedup is almost double on 121 than it is on 31 threads.

D. Breadth-First Search

The speedup achievable using a layered BFS algorithm depends on the structure of the graph. Figure 4(a) and 4(b) shows the performance of the OpenMP-Block and OpenMP-Block-relaxed implementation on two sample graphs from our set, the pwtk and inline_1 graph, respectively. It also shows the maximum achievable speedup according to our performance model presented in Section III-C. For the model, we used as block size the one that yields the best performance in our implementation (32 in this case). Notice that the peak speedup on the inline_1 graph is about twice the speedup achieved on pwtk. Also the slope of the speedup dramatically change at 13 processors on the pwtk graph. This change of slope is explained by our performance model. Moreover it seems to indicate that the margin for improvement on less than 31 threads is quite small on both pwtk and inline_1. After 31 threads, more than 2 threads got scheduled on one core and the ”threads are completely independent” assumption is no longer valid, making the model no longer accurate. As a reference, all the other graphs behave more or less like inline_1: pwtk is an outlier in our experiment.

On the all graphs, the relaxed queue variants led to consistently better speedup than the lock-based variants. So we present only relaxed variants of the two implementation using blocked queues and the algorithm using a bag on all the graphs in Figure 4(c). It appears that the implementation using the bag data structure performs poorly on Intel MIC whereas the implementation based on the blocked queue performs better. The OpenMP-Block-relaxed implementation (using the dynamic scheduling policy) seems to perform slightly better than the model predicted up to 37 threads and then its performance decreases. This tells us, the Intel MIC architecture allowed us to exploit all the parallelism contained in the algorithm. The performance of the TBB-Block-relaxed implementation (using the simple partitioner) is closer to the predicted performance. Notice that none of the implementation we present seem to be able to properly use the SMT capabilities exposed by the architecture.

To understand why the CilkPlus-Bag-relaxed implementation does not scale well on Intel MIC, we executed the different implementations on the host instead on running on the coprocessor. Recall the host has 12 cores and hyperthreading. Figure 4(d) presents the speedup obtained on CPUs and the modeled performance. We also included for comparison the performance obtained by the OpenMP-TLS algorithm (from SNAP) that does not use relaxed queues. On regular, CPUs the Bag and TLS based implementation perform significantly slower than our Block queue implementation (except using 23 and 24 threads where a performance issue in the OpenMP runtime system appears). The bag based implementation was not scaling on CPUs, therefore it comes with no surprise that it performs poorly on Intel MIC.



(a) Using OpenMP (b) Using Cilk (c) Using TBB

Figure 3. Speedup of Irregular Computations on all graphs.

Fairly synchronous algorithm can be executed on Intel MIC even using fine grain parallelism. However complex synchronization schemes induces a large overhead at the granularity making the tradeoff between simplicity and efficiency more difficult to find.

VI. Conclusion

We evaluated the scalability of two graph algorithms on the upcoming Intel MIC architecture using a KNF prototype. Our findings showed that the hardware provided as much parallelism as we could expect. On BFS, the obtained speedup matched a prediction model of the achievable speedup (up to the number of cores). Coloring algorithms showed speedup up to 121 threads. Computationally expensive kernels with irregular dependencies showed speedup of 49.

The Intel MIC coprocessor is programmed using standard shared memory programming paradigm and our programs can easily be executed on both Intel MIC and regular CPUs. We used in our experiment OpenMP, Cilk Plus and TBB. OpenMP is a very well known programming model. Cilk Plus programs are almost written like a sequential programs. And TBB uses a clear object oriented programming paradigm. We managed to achieve higher performance using OpenMP. But it should be noted that the algorithm we implemented exposes very simple parallelism while Cilk Plus and TBB should be able to manage more complex form of parallelism. However when the computation volumes slightly increased, the three programming model yield similar performance.

Our experiments showed that by simply running shared memory codes on a larger system, one should not expect to directly achieve much higher performance. Many design issues are easily overlooked in small shared memory system such as properly exposing the right level of parallelism and limiting the number of atomic operations. Such issues are not necessarily be noticeable on small systems but could dampen the performance at a large scale.

The main highlight of our experiments is the SMT capabilities of the Intel MIC architecture. It allows to achieve linear speedup in memory intensive kernels and is necessary to achieve peak speedup on irregular computationally expensive kernel. Yet SMT is a double-edged sword since any unnecessary operation increases both the sequential runtime and dampens the scalability of the application by increasing in-core pressure.

Overall, we found the Intel MIC architecture to be easy to program and scaled gracefully on challenging graph algorithms. However, we performed all our test on a prototype card and we are looking forward to perform more evaluation on the final design.

Acknowledgment

This work was partially supported by the U.S. Department of Energy SciDAC Grant DE-FC02-06ER2775 and NSF grants CNS-0643969, OCI-0904809 and OCI-0904802.

We would like to thank the Ohio Supercomputing Center for providing us the computational infrastructure; in particular John Eisenlohr and Doug Johnson for their technical support. We also would like to thank Intel for letting us use an Intel MIC prototype and many of its employees for their valuable comments, in particular Paul Besl, Bob Davies, Timothy Prince, James Reinders and Michael Voss for fruitful discussions.

References

[1] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 1, pp. 5–20, 2007.

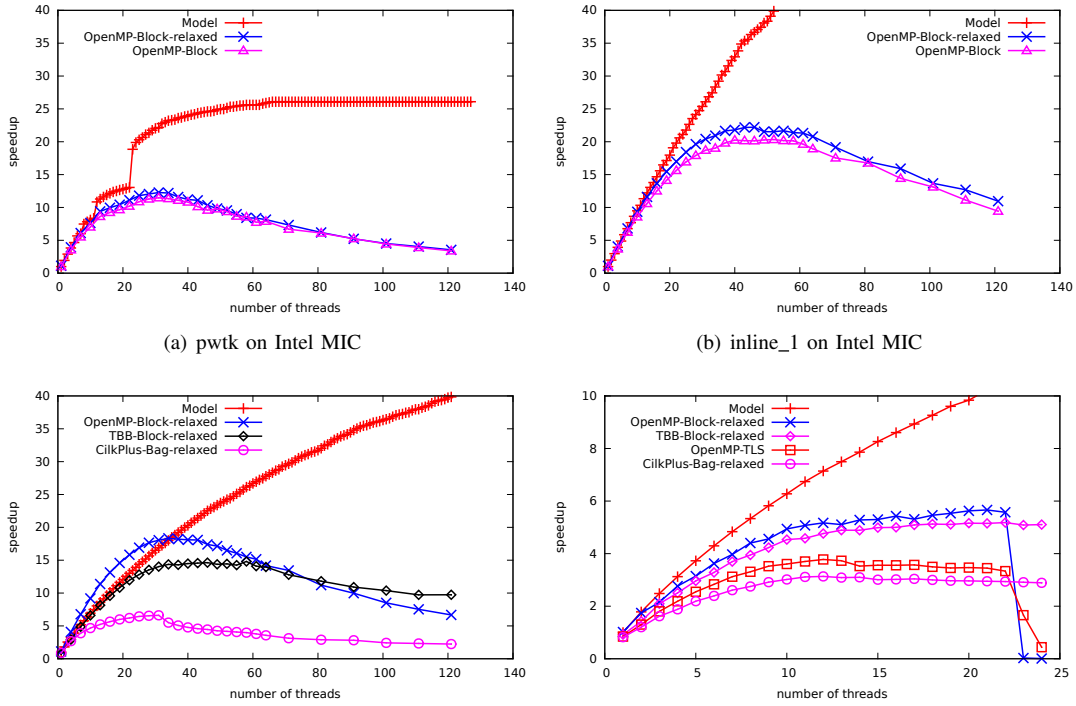
[2] D. Bozdağ, A. Gebremedhin, F. Manne, E. Boman, and Ü. V. Çatalyürek, "A framework for scalable greedy coloring on distributed memory parallel computers," *J. of Parallel and Distributed Computing*, vol. 68, no. 4, pp. 515–535, 2008.

[3] A. H. Gebremedhin, F. Manne, and A. Pothen, "What color is your jacobian? Graph coloring for computing derivatives," *SIAM Review*, vol. 47, no. 4, pp. 629–705, 2005.

[4] J. Allwright, R. Bordawekar, P. D. Coddington, K. Dincer, and C. Martin, "A comparison of parallel graph coloring algorithms," Northeast Parallel Architectures Center at Syracuse University (NPAC), Tech. Rep. SCCS-666, 1994.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. MIT Press, 2001.

[6] U. Brandes, "A faster algorithm for betweenness centrality," *J. of Mathematical Sociology*, vol. 25, pp. 163–177, 2001.



(a) pwtk on Intel MIC

(b) inline_1 on Intel MIC

(c) All graphs on Intel MIC

(d) All graphs on CPU

Figure 4. Speedup of Parallel Layered Breadth-First Search.

[7] M. M. Mathis and D. J. Kerbyson, "A general performance model of structured and unstructured mesh particle transport computations," *J. of Supercomputing*, vol. 34, no. 2, pp. 181–199, 2005.

[8] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1995, pp. 207–216.

[9] M. A. Bender and M. O. Rabin, "Online scheduling of parallel programs on heterogeneous systems with applications to cilk," *Theory Comput. Systems*, vol. 35, pp. 289–304, 2002.

[10] M. Tchiboukdjian, N. Gast, D. Trystram, J.-L. Roch, and J. Bernard, "A tighter analysis of work stealing," in *International Symposium on Algorithms and Computation*, 2010.

[11] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin, "Reducers and other cilk++ hyperobjects," in *Symposium on Parallel Architectures and Algorithms (SPAA)*, Aug. 2009.

[12] D. W. Matula, "A min-max theorem for graphs with application to graph coloring," *SIAM Review*, vol. 10, pp. 481–482, 1968.

[13] D. Zuckerman, "Linear degree extractors and the inapproximability of max clique and chromatic number," *Theory of Computing*, vol. 3, pp. 103–128, 2007.

[14] T. F. Coleman and J. J. More, "Estimation of sparse Jacobian matrices and graph coloring problems," *SIAM J. on Numerical Analysis*, vol. 1, no. 20, pp. 187–209, 1983.

[15] J. C. Culberson, "Iterated greedy graph coloring and the difficulty landscape," University of Alberta, Tech. Rep. TR 92-07, Jun. 1992.

[16] A. H. Gebremedhin and F. Manne, "Scalable parallel graph coloring algorithms," *Concurrency: Practice and Experience*, vol. 12, pp. 1131–1146, 2000.

[17] Ü. V. Çatalyürek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen, "Graph coloring algorithms for multicore and massively multithreaded architectures," *Parallel Computing*, 2012, (to appear).

[18] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A scalable distributed parallel breadth-first search algorithm on BlueGene/L," in *Proc. of Super Computing*, 2005.

[19] D. A. Bader and K. Madduri, "Snap, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks," in *International Symposium on Parallel and Distributed Processing (IPDPS)*, 2008, pp. 1–12.

[20] C. L. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *Symposium on Parallel Architectures and Algorithms (SPAA)*, 2010, pp. 303–314.

[21] M. M. Strout and P. D. Hovland, "Metrics and models for reordering transformations," in *Workshop on Memory System Performance (MSP)*, June 8 2004, pp. 23–34.