Experimental Analysis of SMP Scalability in the Presence of Coherence Traffic and Snoop Filtering

Mayez A. Al-Mouhamed and Khaled A. Daud
Department of Computer Engineering
King Fahd University of Petroleum &Minerals
31261 Dhahran, K.S.A
{mayez, kdaud}@kfupm.edu.sa

***Abstract*— Commodity multi-core SMPs may generate an enormous amount of coherency traffic. However, the impact of coherence traffic and snoop filtering on parallel program scalability has not attracted sufficient attention. We experimentally analyze the shared data access patterns of four typical applications having different memory layout. An OpenMp optimized execution model is derived for each application with emphasis on data dependencies and implied coherence messages. Using an 8-core SMP we present the obtained speedups versus change in the number of cores and problem scale. A discussion of potential limitation on scalability due to the application or SMP is presented. To assess the coherence behavior and its impact on scalability of parallel programs, a synthetic benchmark which alternates the data block ownership among two cores of the same or different processors is presented. It is found that coherence overheads including snoop filtering are responsible of significant limitation on parallel program scalability. For 8-core SMPs, speedup can be reduced by factors of 2.5 and 5 for row-major and column-major access patterns as compared to the use of private data, respectively. A truly parallel coherence protocol implementation is needed to provide truly scalable shared-memory model.**

Keywords- distributed-memory, HPC, parallel programming, performance evaluation and speedup, shared-memory system.

I. Introduction

High-performance computing (HPC) has obtained prominence through advances in electronic and integrated technologies beginning in the 1940s. In the 1970s, vector and parallel computer evolution was on the move and HPC has been realized and has diffused dramatically. In the 80s HPC started being widely used in the sectors of business, industry and science with some limited applications in defense industry. Innovations in hardware architecture, like hyper-threading in Intel processors, hyper-transport links in next-generation AMD processors, multicore silicon in today's high-end microprocessors from IBM, and emerging IBM HPC clustered solutions, make HPC to move into the mainstream of computing [1].

To reduce the complexity of parallel programming, OpenMp [2,3] has been developed as a standard for parallel programming for shared-memory multiprocessors. It uses a set of APIs for shared-memory and distributed-memory parallel processing. Like other APIs such as POSIX and MPI, OpenMp provides a set of directives that extend the sequential program and guide the compiler to parallelizing the code. On-chip multiprocessors show much better performance in parallelizing large problems because of locality and low latency communication. OpenMp can also be used in advanced parallel technology like hyper parallelism.

Building large SMPs [4] is motivated by the need to amortize coherence controller and network interface costs over several processors. The use of larger per-processor L2 caches and faster processor-memory buses has helped to alleviate the bus bandwidth problem in connecting several very-high-speed processors. It is imperative that the coherence be designed to handle the high memory access traffic and the associated coherence traffic that appear on the bus. It is well known that SMPs experience a bottleneck unless the coherence controllers provide high-throughput protocol processing. The results show that using parallelism in coherence controller has significant impact on the performance of applications with high communication requirements.

SMP cache coherence is responsible for substantial overhead as it may reduce available bandwidth due to delays in shared memory accesses. Data ownership model implies that only one cache can have data in the "dirty" state. Any write to a cache line that is not "owned" results in a broadcast to all other caches to determine if the cache line is "owned" elsewhere, i.e. induces a look-up in the data cache of all remote CPUs in-order to check data consistency. Actually, the owner notifies the rest of the caches and gives up ownership to the cache that initiated the request. About 90% of these look-ups result in a data cache miss which means the searched line is not found in the local caches [5].

Methods for filtering the majority of snoops that would not find a locally cached copy were proposed. Most of these proposals are based on a family of simple filter mechanisms that dynamically detect most non-shared regions [6]. It defines a region to be a continuous, aligned memory area whose size is a power of two. This allows determining that many requests find that no other node caches a block in the same region even for regions as large as 16K bytes. In [5] demonstrate that the ability to filter 74% (average) of all snoop-induced tag accesses between the bus and L2 that miss in the local caches. This result in an energy reduction of 29% measured as a fraction of the energy required by all L2 accesses for both tag and data arrays.

The Blue Gene/P uses a destination-based snoop filter that combines stream registers and snoop caches to capture both the locality of snoop addresses and their streaming behavior [7, 8]. This allows the snoop filtering to squash snoop requests for addresses known not to be in cache improves performance significantly for caches that cannot perform normal load and snoop lookups simultaneously. Significant reduction in snoop lookups is reported which yields power savings. However, the scientific applications that Blue Gene/P can handle avoid sharing between the cores. As a result, most of the invalidations applied to the caches are useless and could be eliminated.

The massive-multicore [9, 10] is expected to have 1000 or more cores in a few years. To maintain scalability, rising core counts mean higher memory access rates. These bandwidth limitations require that more of the data be stored on-chip using large shared last-level on-chip caches. This approach does not scale beyond a few cores, while its power requirements grow in a quadratic manner with size, which exclude their use in chips on the scale of 100s of cores.

Our objective is to analyze limitations on scalability in parallel programs on an SMP having two quad-core processors, i.e. 8 cores. We review the main OpenMp parallel constructs and analyze the organization, recurrences, and data access patterns for deriving the OpenMp execution model for a set of 4 applications which are selected due to their specific and frequent change coherence traffic patterns. These computations are: Jacobi solving linear system equations, ocean simulation, bucket sorting, and alternating direction integration. For each application we derive the OpenMp execution model and determine the sequence of coherence operations. We present speedup analysis versus change in application and processor scales and discuss potential limitation on parallel scalability due to coherence traffic. We also present a small benchmark to assess the scalability of the coherence protocol and snoop filter when the data producers and consumers are spread over the cores and the implied degradation in execution times. Finally we relate the application scalability issues with those observed from the benchmark. These factors can seriously limit the parallel program scalability in SMPs.

This paper is organized as follows. In section 2 some background on Intel x3550 SMP coherence and snoop filter. In section 3 we present the execution models for 4 typical computations. In section 4 we present a benchmark for testing the coherence protocol. Evaluation is presented in Section 5. We conclude in Section 6.

I. BACKGROUND

The Intel x3550 SMP consists of a dual-processor each is a 2.0 GHz Xeon Quad-core (E5405) processors [11], i.e. a total of eight cores. Each core has 32 KB L1 data cache and 32 KB instruction cache. The L2 cache consists of a 2x6 MB shared memory for intra-processor communication. Inter-processor communication uses a 1.333 GHz FSB bus. The MESI coherency protocol is used for the 8 core caches. MESI is implemented as a central coherency engine located in the memory controller (chipset). A snoop filter (SF) which stores tags and coherency state information for all caches in both processors, using a directory scheme. SF keeps track of which cache lines are where and filters coherence traffic. The filter is organized as two bus interleaves; each contains 8K sets which are arranged as a 16-way set associative array. This allows tracking of up to 256K cache lines. When the coherency engine intercepts a request from the front-side bus, a snoop filter lookup is performed to determine a hit or miss. The state of an existing entry is then updated or a new entry is allocated. Depending on the search results of the snoop filter, the coherency engine will make a decision as to whether or not a cross-buss snoop will be launched. The snoop filter significantly reduces the number of snoops each core-cache must perform following a cache access.

It is clear from the above that cache misses can dramatically impact the data sharing latency whenever the data producer is not on the same core as the consumer. It is useful to assess the impact on performance due to some frequent access pattern: a sub-array is written by some core and later being used in another core. While many studies have been conducted to reduce coherence overheads to reduce power consumption in SMPs little effort has been made to analyze the impact on scalability of the coherence protocol with or without snoop filtering.

II. OPTIMIZED EXECUTION MODEL

In this section we analyze four typical computational programs [12]. These are: Jacobi solving linear system equation, the Ocean Simulation red-black, Bucket Sorting, and the Alternating Direction Integration (ADI) [13]. These computations are analyzed respect to their organization, the presence of recurrences, and data access pattern. For each case we derive an OpenMp optimized execution model that can be easily implemented after taking advantage of the advanced parallelization constructs of OpenMp. In the following we describe the execution model of each of these computations.

A. Jacobi Solver

Jacobi solving linear system equation consists of iteratively solving a linear system equation Y = AX which consists of n linear equations with n unknowns, where X is the unknown, Y is a given nx1 vector, and A is a given nxn matrix. In the kth iteration, the kth unknown is computed based on the knowledge of the global solution from the last iteration. The process is initialized with the solution X=Y. The solution equation can be used as an iteration formula for each of the unknowns to obtain a better approximation. Following the cooperative computing of a solution in local storage, the new solution is copied into the global solution in shared memory before iterating the process. Note that updated values of X in a given iteration should not be used by other workers in the same iteration.

The OpenMp execution model for Jacobi is shown on Fig 1-(a). At each step, each core reads the solution X that was cooperatively computed in just the previous iteration by the p cores. Each core potentially generates (1-1/p)n read misses for data computed by other cores, i.e. data being in the invalid state. There are 2 operations following each R_{MISS} which is completed by a WB. In response to the above, each core contributes in n/p write-backs and change block state to shared. However, the read misses will be served by n write-backs if no more than one read-miss is broadcast to all cores for each distinct data element.

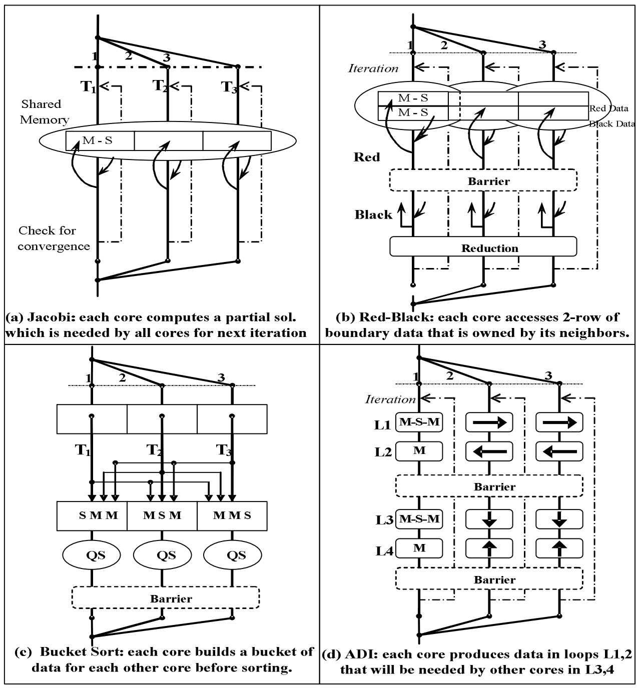
After a complete solution X is acquired by each core through above WBs, each core computes n/p partial results in private memory without coherence traffic. A barrier ensures all cores completed. Now each core copies n/p partial results onto shared memory which leads to generation of n/p bus invalidations. These are processed by all the p-1 caches which are required by the algorithm access pattern as the solution will be needed in the next iteration by all cores. The core operations involving the sequencing of coherence operations with arithmetic operations can be described for each step by following short notation:

JACOBI: core{n(1-1/p)(R_{MISS}/WB, 2), n/p(-,2n), n/p(Inv,-)}

In total we have nk R_{MISS}/WB and nk Invalidations for k iterations.

B. *Ocean Simulation*

The Ocean Simulation has one known implementation called the Iterative Gauss-Seidel linear equation solver. It consists of a set of near-neighbor sweeps (iterations) to convergence: (1) interior n-by-n points of (n+2)-by-(n+2) updated in each sweep (iteration), (2) updates done in-place in grid, and difference from previous value is computed, (3) accumulate partial differences into a global difference at the end of every sweep or iteration, (4) check if error (global difference) has converged to within a tolerance parameter, and (5) If so, exit solver; if not, do another sweep (iteration) or iterate for a set maximum number of iterations. Due to data dependencies the concurrency is along the anti-diagonals which require the use of too many point-to-point synchronizations. One approximation is to restructure the loops (simply ignore dependencies) and reorder the grid traversal into a red and a black ordering. Now the red sweep and black sweep are each fully parallel. A conservative but convenient global synchronization is needed between the above the red and black loops. The OpenMp execution model for the Ocean Red-Black is shown on Fig. 1-(b). In the red-loop the black data is read to compute the red points. The coherence status changes from modified to shared for



Shared Memory
M - S
Check for convergence
(a) Jacobi: each core computes a partial sol. which is needed by all cores for next iteration
Iteration
Red Data
Black Data
Red
Barrier
Black
Reduction
(b) Red-Black: each core accesses 2-row of boundary data that is owned by its neighbors.
S M M
M S M
M M S
QS
Barrier
(c) Bucket Sort: each core builds a bucket of data for each other core before sorting.
L1
L2
L3
L4
M-S-M
M
(d) ADI: each core produces data in loops L1,2 that will be needed by other cores in L3,4

Fig 1: OpenMp execution models for Jacobi, Ocean Red-Black, Bucket sorting, and ADI.

App.	Core Bus Transactions	Rm/WB	Inv
Jacobi	{n(1-1/p)(R_{MISS}/WB, 2), n/p(-,2n), n/p(Inv,-)}	nk	nk
Ocean	{n/2(R_{MISS}/WB, 4,Inv), (n/p-2)n/2(-,5), n/2(R_{MISS}/WB, 4,Inv)}	2npk	2npk
Bucket	{n/p(E,-), n/p(1-1/p)(-), n/p(1-1/p)(Inv,-), n/p(1-1/p)(R_{MISS}/WB,-), Quicksort(-), n/p(1-1/p)(Inv,-)}	n(1-1/p)	2n(1-1/p)
ADI	L1:[n^2/p^2(5), n^2(p-1)/p^2(R_{MISS}/WB,4,Inv)],	2 n^2(p-1)k/p	2 n^2(p-1)k/p
	L2:[n2/p(5)],		
	L3=L1,		
	L4=L2.		

Table 1: Sequence of bus transactions and operations

black data, and to modified for red data. The barrier is needed because of data dependencies over boundary elements across distinct threads. The reduction is needed to evaluate a global convergence parameter. For distributed memory, only boundary data elements may cause page faults when a process reads data that is remote, e.g. near-neighbor data dependence.

Here a block-row partitioning is used and each core has a fixed range of data. The algorithm consists of the red and black loops and iterating again. In the red loop (black loop), each core reads $n^2/2p$ array elements which are black-points (red-points). The $n^2/2p$ reads are previously in the Modified state (owned) except first and last rows (2n) which are in the invalid state. The sequencing of coherence operations and arithmetic operations can be described by following short notation:

OCEAN: {n/2(R_{MISS}/WB, 4, Inv), (n/p-2)n/2(-,5), n/2(R_{MISS}/WB, 4, Inv)}

In total we have 2npk R_{MISS}/WB and 2npk Invalidations for k iterations.

C. *Bucket Sorting*

The parallel Bucket Sorting (PBS) algorithm aims at sorting n data elements. PBS is a typical example of Divide-and-Conquer paradigm. In a sequential version it will require n steps to place n numbers into p buckets, where p is the number of processors or threads. If sorted data is uniformly distributed then each bucket will have approximately n/p numbers. In other words, the data is partitioned and each thread clusters its range of data into all possible buckets. Next the numbers in each bucket (thread) must be sorted, i.e. each thread must sort its own aggregate cluster. In other words, sequential sorting algorithms can be used within each thread like Quicksort or Mergesort. Finally the aggregation of the partially (by each thread) sorted lists represents the overall sorted data. The OpenMp execution model for the PBS is shown on Fig. 1-(c). Each thread operates over its range of data and clusters it into all possible p buckets, i.e. whose status becomes exclusive for each core. Next each thread sorts its aggregate list of buckets causing a status change from private shared. This produces a sorted partial list. The aggregation of all lists represents the sorted array.

Each core exclusively accesses n/p elements (E state) in its range and cluster the data into p small buckets in private memory. Next, each core copies u = n/p(1-1/p) elements from its small buckets onto a shared array. This causes the broadcast of u invalidations which are processed by all caches. However, the data is used later in only one destination cache. To sort data in its range, each core reads miss u elements from its small buckets, which cause u write-backs by each core. The WBs from all cores are all distinct which leads to n(1-1/p) aggregate WBs. To sort up its range of data, each core uses Quicksort in private memory and updates the shared array with the sorted data which causes u invalidates. The sequencing of coherence operations and arithmetic operations can be described by following short notation:

Bucket: Cores{n/p(E,-), n/p(1-1/p)(-), n/p(1-1/p)(Inv,-), n/p(1-1/p)(R_{MISS}/WB,-), Quicksort(-), n/p(1-1/p)(Inv,-)}

In total there are n(1-1/p) R_{MISS}/WB and 2n(1-1/p) Invalidations.

D. *Alternating Direction Integration*

The Alternating Direction Integration (ADI) [12] program is a small piece of code with its structure and recurrences represent a scientific iterative algorithm. ADI consists of repeatedly executing a set four loops: (1) L1 is a left-right forward sweep, (2) L2 is a right-left backward sweep, (3) L3 is an up-down sweep, and (4) L4 is a bottom-up sweep. Each loop has two recurrences (arrays x and b) along its sweep direction, i.e. reads a data that was written in the previous iteration. The code for L1 is:

```
for (i = 0; i < N; i++)              /* Loop 1 from ADI
{    for (j = 1; j < N; j++)
       {  x[i][j]=x[i][j]-x[i][j-1]*a[i][j]/b[i][j-1];
           b[i][j]= b[i][j] - a[i][j]*a[i][j]/b[i][j-1];
       }
}
          x[i][N-1] = x[i][N-1]/b[i][N-1];
}
```

Block data partitioning along the rows avoids crossing the recurrence while running L1 and L2 which consists of horizontal sweeps. Similarly, block data partitioning along the columns avoids crossing the recurrence while running L3 and L4 which consists of vertical sweeps. The OpenMp execution model for ADI is shown on Fig. 1-(d). In each step, each data block (i,j) alternates its modified status from core I to cores j and vice versa. In loop L1, the block (i,j) is read and becomes first shared and later written which makes it modified by core i. L2 uses the same row-partitioning as L1 keeping the same status (modified) because each block is read and modified by the same core. In L3 a column partitioning is used, each core j reads the data (i,j) and then modifies it. This leads to change the status from being modified by core I to shared and then to modified by core j.

While running L1, each core has a block-row partition (n^2/p). Each core makes u=n^2(p-1)/p^2 read misses for each data array. Overall there are pu read misses on the bus, where each read miss is served by a distinct data write back that is exclusively used by one single core. Similarly, each core makes u writes for each of the two arrays which cause the broadcast of u invalidates. Although the invalidation is sent to all caches, each array element is being used by just two cores in ADI.

Overall there O(n^2) read misses sent to all caches, O(n^2) write backs whose data is used by a single core, and O(n^2) invalidations to serve one single core but processed by all caches. Note that a similar access pattern repeats for each of the four loops. The sequencing of coherence operations and arithmetic operations can be described by following short notation:

ADI: { L1:[n^2/p^2(5), n^2(p-1)/p^2(RMISS/WB,4,Inv)], L2:[n^2/p(5)], L3=L1, L4=L2}

This leads to u=2n^2(p-1)k/p R_{MISS}/WB and u Invalidations.

The Omp parallel construct is used to avoid repeated thread entry and exit with the four loops and the outer iteration loop. The iterative loop is embedded within each thread, e.g. each thread repeats for all iterations. No barrier is used between L1 and L2 as well as between L3 and L4 as the data produced in L1 is read in L2, e.g. producer-consumer falls within each thread local data. A barrier is

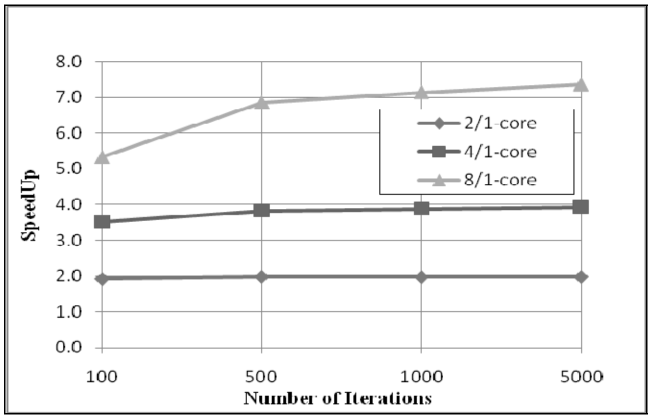


Figure 2: Speedup for Parallel Jacobi (500 Equations).

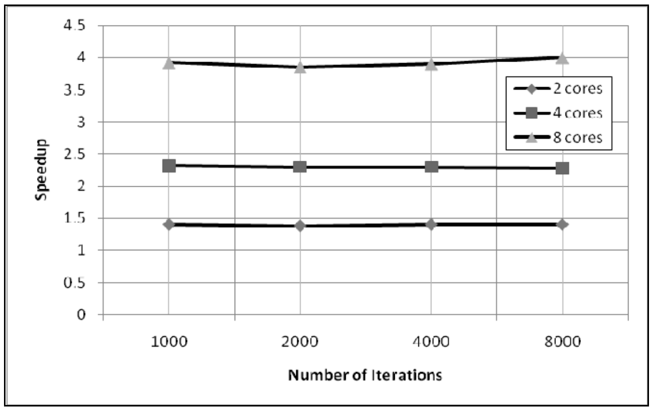


Figure 4: Speedup of Parallel ADI (400x400)

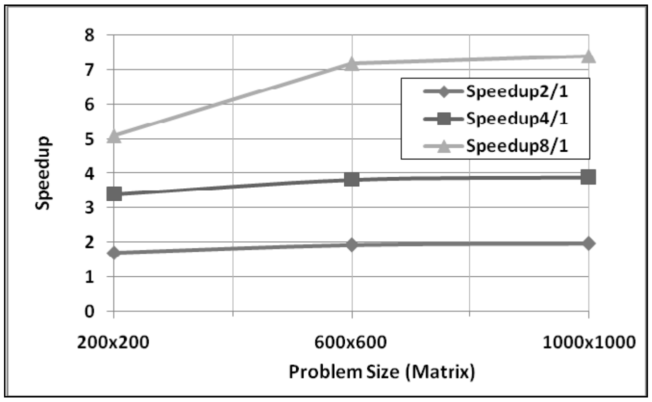


Figure 3: Speedup of Ocean Red-Black

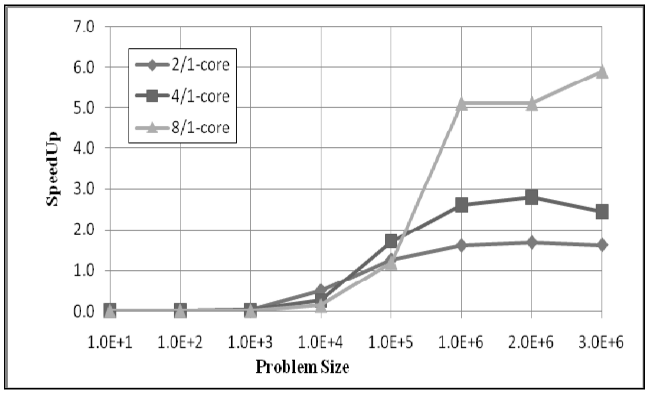


Figure 5: Speedup of Parallel Bucket Sort

needed between L2 and L3 as well when iterating from L4 back to L1. In this case, there are data dependencies across the data produced by each thread.

I. Evaluation

Here we run the parallel programs that derive from the previously presented execution models. We present speedup performance analysis and discuss pros and cons.

The Parallel Jacobi Algorithm (PJA) was tested for NxN arrays with varying values of N. Fig 2 shows the speedup versus scaling the parallel machine size (number of cores) and the number of iterations of PJA, which is reported on the X axis, with a problem size of N=500. Each node in the e1350 cluster is a 2-processor and each processor is a quad core. The PJA speedup scales very well within the shared-memory of a processor (up to 4 cores). Small problem sizes (case of 100 iterations over 8 cores or 2 processors) suffer from additional overhead across the two processors within a single node. PJA with large iteration space overcomes the above limitations and provide near-linear speedup in all cases.

Fig 3 shows the speedup obtained out for the Parallel Ocean Simulation (POS) (Gauss-Seidel). The speedup is plotted versus the number of cores (2, 4, and 8) and the problem size. Here, the number of iterations is controlled by the needed tolerance. We obtain near linear speedup only for large problem size. Analysis of the run time using OpenMp Profiler ompP (Version 0.7.0) shown on indicates that the barriers at end the red and black loops incurs only about 6% of total thread time. For this, it is believed that the limitation on scalability of small problems is due to the reduction at the end of each step.

Fig 4 shows the speedup obtained out for the ADI execution model. The speedup is plotted versus the number of cores (2, 4, and 8) and the number of iterations. Disregarding the problem scaling, it appears that the main limitation of speedup is due to node issues. Linear speedup is obtained (not shown) when running ADI with only loops L1 and L2 which implicitly uses a block-row partitioning. However, a sub-linear speedup is observed for ADI with only loops L3 and L4 which must use a block-column data partitioning. The reason for the above is the row-major memory allocation which favors block-row portioning and presents a limitation when the problem requires block-column data partitioning. To fix the problem, we use a matrix transpose between loops L1-2 and L3-4 which enables the use of block-row partitioning for L3-4.

Fig 5 shows the speedup obtained for parallel bucket sort (PBS) versus scaling the parallel machine (cores) and the problem size (up to 3 million elements) which is reported on the X axis. The speedup is very low for small problem size because of the overhead in the aggregation of small buckets into a large bucket (copying) of each thread, e.g. overhead dominates the thread work. The PBS is not scalable for small problems, but for large problem size a near linear speedup is obtained for 2, 4, or 8 cores. The correctness of PBS is confirmed by comparing its result to that of the sequential solution.

II. TESTING THE COHERENCE PROTOCOL

To assess the effect of switching ownership between the cores of some processors or different processors we used a test program (TP). The program consists of iterating between the previously described loops: L1 (forward sweep) and another loop L2. L1 is partitioned among the cores using a block-row distribution, i.e. core (C_i) is allocated the ith block of rows (R_i). This relation is denoted by (C_i - R_i). In loop L2 the mapping is done using a block permutation defined by $C_i – R_j$, where j=i+k mod 4, for k=0-3.

By varying k, ($TP_{ROW-INTRA}$) implements four possible block-row ownership swapping among the 4 cores of each processor as shown on upper group of permutations of Fig. 7. The column numbers from 0 to 7 in the above figure denotes the SMP cores and the lines between two columns of numbers denote the swapping of block-row ownership from L1 to L2, i.e. read and modify data. $TP_{ROW-INTRA}$ allows assessing the protocol overhead due to swapping block-row ownership at the level-2 shared-cache within each processor. TP can also be used to implement ($TP_{ROW-INTER}$) block-row ownership swapping among all the cores of both processors when j=i+k mod 8, for k=0,4. This allows covering the straight permutation (k=0) as well as all other interesting possible permutations (k=1-4) as shown on lower group of permutations of Fig. 7. When k=1, each core of a given processor owns a block-row that was owned by a core from the other processor. Each of the 3 cores of the other processor owns a block-row that was owned by a core of the same processor. This denotes by 2(1,3) each of two processors are swapping 1 block row from each another (inter processor) and 3 block-row from within each processor. $TP_{ROW-INTER}$ allows assessing the protocol overhead due to swapping block-row ownership at the shared-bus snoop filter system prior to access the shared-memory among the two processors.

To assess the effects of block-column partitioning, we use a similar TP test program but with modified L1 and L1' to implement a block-column distribution. This allows to asses block-column ownership permutations among the 4

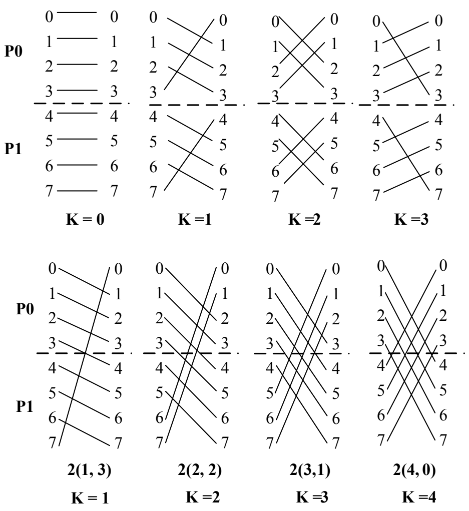


Figure 6: Swapping block-row ownership: (upper) among 4 cores of each processor and (lower) among 8 cores of both processors.

cores of each processor apart ($TP_{COL-INTRA}$) and among all 8 cores ($TP_{COL-INTER}$).

Fig. 7, 8, and 9 show the running of four (scenarios) test programs $TP_{ROW-INTRA}$, $TP_{ROW-INTER}$, $TP_{COL-INTRA}$, and $TP_{COL-INTER}$ for arrays of size 240x240, 480x480, and 720x720 (largest possible), respectively. Each figure shows the plot of execution time of TP for each of the four scenarios versus the value of permutation parameter k.

The above test programs were run with and without the recurrence. The recurrence exposes the compiler capability to hide the latency due to the recurrence among the successive iteration (see code of L1), i.e. the RAW that takes place from iteration to another due to references x[i][j], x[i][j-1], b[i][j], b[i][j], b[i][j-1]. It is obseved that TP has been speeded up by about 10% to 15% when the 2 recurrences have been removed from the code (not shown).

A. *Scalability with data size*

The processed data size shown in Fig. 8 (N=480) and Fig. 9 (N=720) is four times and nine times compared to the data processed in the experiment shown on Fig. 7. However the execution time of TP shown in Fig. 8 and Fig. 9 is 2.5-3.0 times and 5-7 times compared to that shown in Fig. 7. Based on the above it is clear that TP execution time is sub-linear with the size of processed data.

B. *Block-row versus block-column distribution*

The analysis of results displayed on Fig. 7-9 show that the execution time of TP is dominated by whether the block-row or block-column data distribution is used. The block-column distribution is responsible of no less than 60% to 100% degradation as compared to a block-row distribution. The highest degradation is observed for the smaller data size. These degradation are due to cache data reuse which is

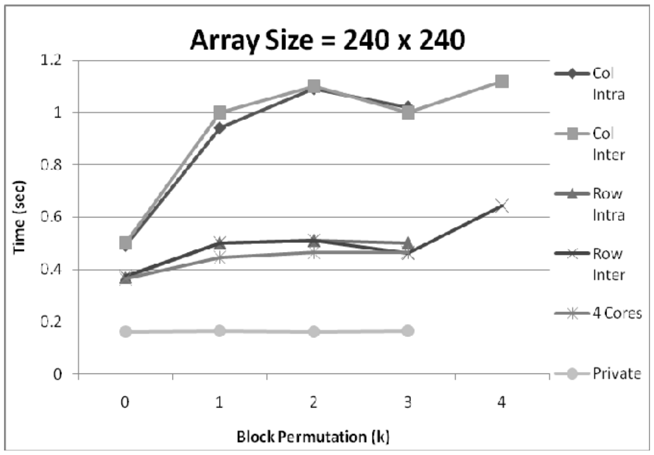


Figure 7: TP parallel times versus block ownership permutations for 240x240 arrays.

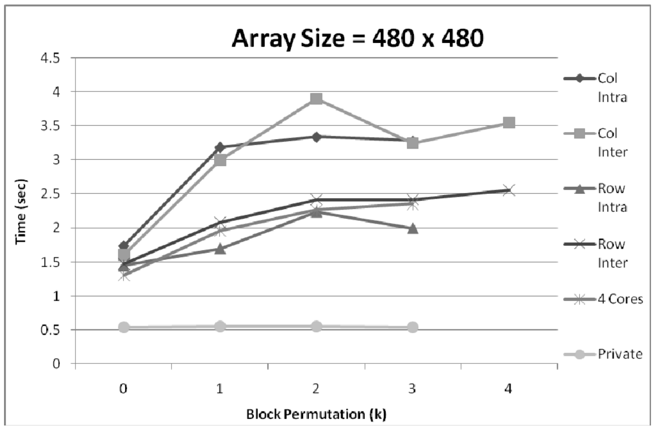


Figure 8: TP parallel times versus block ownership permutations for 480x480 arrays.

poor in the case of block-column distribution, i.e. cached block are stored according to row-major storage. Applications in which the data access alternates between block-row and block-column (like ADI) have noticeable limitations due to the inneficiency of data access under block-column distribution.

C. *Reference case: all snoops are filtered by SF*

Fig. 7-9 show the effects of swapping the block ownership among the cores of each processor (Intra) or both processors (inter). The above Intra and Inter block swappings produce degradation on the execution time due to the implied cohence traffic. The reference case is when the cores read and write to the same pages they owned in both loops L1 and L2, i.e. the straight mapping shown on Fig 6 for k=0. In this case, the coherence notices are forwarded to the FSB bus which interfaces the level-2 cache to the share-memory (DRAM). These notices are filtered by the snoop

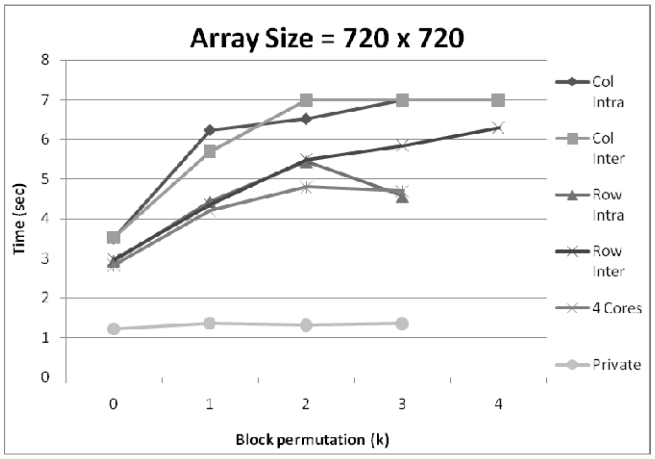


Figure 9: TP parallel times versus block ownership permutations for 720x720 arrays.

filter because the referenced blocks are not used by the other caches. Therefore, the reference case corresponds to TP execution without having cohenrece notices reaching the other caches.

D. *Intra-processor vs. inter-perocessor data permutations*

The block permutations shown on Fig. 6 in the upper group (Intra) and lower group (inter) when k is varied between 1 and 4 causes the swapping of page ownership. In this case, significant degradation in execution time (between 60% to 100%) as compared to the reference case appears when N is 480 and 720, respectively. When each core is running a given loop (L1 or L2), it reads and writes to a page that was owned by another core in the previous loop (L2 or L1) due to iterative nature of TP. It is clear that swapping of page ownership exposes the latency of the coherence protocol including the snoop filtering. The reason is that the reads and writes operations which are carried out by a given core produce snoops that must be processed by all the cores because each page is cached in exactly two cores (Fig. 6) when iterating over L1 and L2. The snoop filtering aims at reducing the snoops for the cores that will produce misses for these snoops, i.e. for the cores that have no copies of the accessed pages. The data access pattern in L1 or L2 consists of following there steps:

1. Core C_i reads the shared data which creates read misses (block was in "Invalid" state) that propagates to all caches of this processor as well as to the FSB bus which interfaces the main memory.
2. Since SF acts the FSB level, the snoops are forwarded to all caches of the other processor.
3. The owner core (block in "modified" state) on the same processor or on the other processor will write back the block and make it shared.
4. Many reads by C_i cause no soonps.

5. Later, C_i writes the same block which causes an invalidation to propagate to all caches through SF and be processed in C_j identified by j=permutation(i). The block becomes modified in C_i and invalid in C_j.

The above process repeats for all accessed blocks because each iteration of TP consists of L1 and L2. The overhead is introduced when the status of a page changes from "modified" to (1) "shared" with a write back and then to "modified" with an invalidation. It is clear that the protocol and the snoop filtering are responsible for the degradation caused by normal snoop transactions which significantly limit the program scalability in TP. It is clear that the snoop filter between the FSB and shared-memory bus which interfaces the shared level-2 cache to the share-memory (DRAM) has dominant overhead on parallel execution.

In addition, Fig. 7-9 show that there is a marginal difference in performance when the page permutations are carried out within a processor (intra) or across both processors (inter). It is shought that the above should be attributed to the presence of the snoop filter between the FSB and shared-main memory. Thus all snoop traffic corresponding to normal protocol processing of sharing of pages passes through the snoop filter before being transmitted to all the cores disregarding whether these cores are within the processor or on the other processor. At this level, the protocol implementation does not lead to noticeable difference in performnce when block ownership swapping is "Intra" or "Inter".

III. CONCLUSION

We analyzed the organization, recurrences, and data access patterns of four typical computational programs which are: solving linear system equation, ocean simulation, bucket sorting, and the alternating direction integration. For each case, we derived an OpenMp execution model to ease the design of the parallel algorithm. In the evaluation we run the above algorithms on an 8-core SMP and plotted the speedup versus scaling both of the machine and problem size. A run-time profiler tool was used to assess thread time, overhead, and barriers. Plots of the speedup were discussed and potential limitations in scalability were pointed out in connection with the coherence protocol, snoop filtering, and block ownership bus transactions. A benchmark has been developed to focus on the producer-consumer (block ownership) relationships across the cores of each processor and across cores of both processors. Analysis of potential limitations on scalability is presented versus loop recurrence, row-major and column major storages, and possible permutations in the data producer-consumer among the SMP 8 cores. It is found that coherence overheads including snoop filtering are responsible of significant limitation on scalability especially for applications where data producer and consumer are on distinct cores. In this case, speedup can be reduced by factors of 2.5 and 5 when using row-major and column-major storages for 8-core SMP as compared to the use of private data, respectively. More research is needed to developing truly scalable coherence protocols which are the pre-requisite for scalable SMPs.

ACKNOWLEDGMENT

The authors would like to thank King Fahd University of Petroleum and Minerals (KFUPM) for the computing support.

REFERENCES

[1] L. T. Yang, M. Guo, "High-Performance Computing: Paradigm and Infrastructure", L. T. Yang, M. Guo, Wiley, December 2005.

[2] M. Sato, "OpenMp: Parallel programming API for Shared Memory Multiprocessors and On-chip Multiprocessors", Proc. of Intl. Symp. on Sys. Synthesis, pp. 109-111, 2002.

[3] Ayon Basumallik,Seung-Jai Min and Rudolf Eigenmann. "Programming Distributed Memory Sytems Using OpenMp". IEEE Computer, 2007.

[4] Nanda, A. K.; Nguyen, A.-T.; Michael, M. M.; Joseph, D. J.; , "High-throughput coherence control and hardware messaging in Everest," *IBM Journal of Research and Development* , vol.45, no.2, pp.229-243, March 2001

[5] Moshovos, A.; Memik, G.; Falsafi, B.; Choudhary, A.; JETTY: filtering snoops for reduced energy consumption in SMP servers, Inter. Symp. on High-Performance Computer Architecture (HPCA), 2001 , pp. 85 – 96.

[6] Moshovos, A.; RegionScout: exploiting coarse grain sharing in snoop-based coherence, Inter. Symp. on Computer Architecture, 2005. ISCA, 2005 , pp. 234 – 245.

[7] Blocksome, M. et al.; Design and Implementation of a One-Sided Communication Interface for the IBM eServer Blue Gene, Proceedings of the ACM/IEEE SC 2006 Conference, 2006 , Page(s): 54-58.

[8] Salapura, V. ; Blumrich, M. ; Gara, A. ; Design and implementation of the blue gene/P snoop filter, High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on, 2008, pp. 5.

[9] Lis, Mieszko; Shim, Keun Sup; Cho, Myong Hyon; Devadas, Srinivas; , "Memory coherence in the age of multicores," *Computer Design (ICCD), 2011 IEEE 29th International Conference on* , vol., no., pp.1-8, 9-12 Oct. 2011

[10] Montaner, H.; Silla, F.; Fröning, H.; Duato, J.; , "Getting Rid of Coherency Overhead for Memory-Hungry Applications," *Cluster Computing (CLUSTER), 2010 IEEE International Conference on* , vol., no., pp.48-57, 20-24 Sept. 2010

[11] Radhakrishnan, S.; Chinthamani, S.; Kai Cheng; , "The Blackford Northbridge Chipset for the Intel 5000," Micro, IEEE , vol.27, no.2, pp.22-33, March-April 2007

[12] Parallel Computer Architecture (PCA): A Hardware/Software Approach, David E. Culler, Jaswinder P. Singh, Morgan Kaufmann Publishers, 1999

[13] K. Kennedy and U. Kremer, Automatic Data Layout for Distributed-Memory Machines, ACM Trans. Program. Lang. Syst., 20:4, 1998, pp. 869-916.