CS458 Software Coherence

For

Disciplined Parallelism

Xi Jin Yang Song

## 1 Introduction

In the conventional small-scale share-memory system, hardware snooping protocol can be used to maintain cache coherence. When the scale is increasing, snooping protocol creates great burden on the bus since snooping protocol requires broadcast messages to communicate. Hardware directory-based coherence protocol can be applied that the information of ownership and state for each cache line is maintained in directory and every message is passing between two nodes which releasing the pressure from broadcast style. The split-transaction bus can meet the requirement. However directory protocol increases hardware design complexity and brings in additional meta-data storage cost for maintaining sharing pattern bit. For instance, maintaining a 1024-core share memory system requires 1024 presence bit for every cache. Assume we have 128KB 16ways L1 cache with 64B block size, we have 1024 cache lines and directory need 2MB to record presence bit only. Another disadvantage of hardware-managed coherence is that people cannot do anything with the machine shipped. However software-managed cache coherence provides more scalability and it’s more attractive because the overhead of detecting conflicting data is moving from run-time to compile time. The basic idea is that we can move cache coherence responsibility from low-level hardware to compiler, high level language and programmer.

Deterministic Parallel Java is an interesting topic. It is an extension to Java that guarantee there is no data race within paralleled loops by enforcing deterministic-by-default semantics via compile-time type checking. DPJ guarantees the parallel program has a sequential ordering which implies a read should always return the value of the last conflicting write before it in the sequential program order. Since there is no data race, there is no other writer or reader for one cache line in a parallel phase. It’s not necessary to keep maintaining directory shadow tag and sharing vector because there is always at most one writer or one reader of a cache line at any phase. Also write invalidation traffic can be eliminated. A new software coherence schema called DeNovo will be introduced at the end of this report.

Foreach()

{

//parallel

}

## 2 Deterministic Parallel Java

Many features are introduced into Java:

1 Fork-Join Parallelism

Foreach()

{

//parallel

}

--pthread\_create

--barrier

--pthead\_join

DPJ frees the programmer from thinking of how to create threads and manipulate them such like synchronization and join. In DPJ, a runtime will be provided to support fork-join parallelism.

2 Regions:

Programmer should assign different region name to all the objects. Basically, it’s marking different memory location with regions. DPJ will check the determinism according to the region accesses.

3 Effect Summary:

Programmer should annotate methods with “effect summary” by marking the region they want to read and write. Wrong summary may lead to compiling error. So programmer should know exactly what the data structure like, how they organize and which region they want to operate on.

The first step we do is to understand how red-black SOR works in terms of data structure, algorithm and cache operations. Measure the speedup and synchronization cost on pthread-based C program.

The second step is we migrate all the work to Java and DPJ platform. Measure the performance of DPJ programming and compare with conventional Java program.

Then we develop DPJ-based Gaussian elimination program which is a more sophisticated program for us to study.

With the help of deterministic data-race-free parallel java , the last step is to learn the basic idea of DeNovo – a software coherence schema with simply hardware support.

## 3 Measure a modified red-black SOR pthread-based C program

I add omega and convergence detection code to the SOR program provided by CS458 to make it more practical. Since we have already done this in previous assignments, there will be a simply demonstration here.

**Setting:**

Solving a 1002x402 array by using Red-Black SOR.

omega = 1 relaxation factor that influence convergence speed

EPS = 0.1 This array will converge after 72840 iterations.

Red-Black Fashion to reduce false-sharing and directory traffic:

Red array stored in different memory space from black’s

| R(0,0) | B(0,1) | R(0,2) | B(0,3) | <-- core 1 works on this line

| B(1,0) | R(1,1) | B(1,2) | R(1,3) | <-- core 2 works on this line

| R(2,0) | B(2,1) | R(2,2) | B(2,3) | <-- core 3 works on this line

| B(3,0) | R(3,1) | B(3,2) | R(3,3) | <-- core 4 works on this line

When all cores update red elements, they will request Read for all black elements once and request Read.Ex once for the red elements they are working on. Therefore black nodes array is always in share state in everyone’s cache within this phase and there is no false sharing problem. A barrier will be reached and then red array will be in share state and send Read.Ex for every black node.

**Machine1:**

2 Intel Xeon E5649 Westmere, 2.53GHz, 6-Cores x 2-"Threads"

192 KB (code) / 192 KB (data) L1 4-way, 256KB 8-way L2 Cache and 12MB 16-way L3 Cache

In the test, I only use 6 cores on one chip and skip all the HT thread to keep result accurate. From the observation, socket to socket delay may influence our Sync/total measurement, so I only use 6 physical cores on one socket.

**Convergence Criteria**



In our 2D-dimension:

black\_[j][k] = W\*(red\_[j-1][k] + red\_[j+1][k] + red\_[j][k] + red\_[j][k+1])/4 + (1-W)\*black\_[j][k]

The spectral radius of R is , where the maximum is taken over all eigenvalues of R.

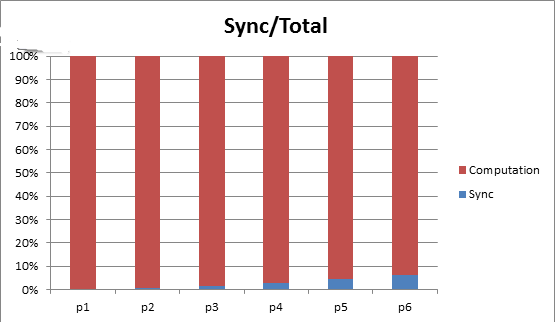
THEOREM 1 Therefore 0 < w < 2 is required for convergence

THEOREM 2 if A is symmetric positive definite, then for all 0 <w < 2,

So SOR(w) converges for all 0 <w < 2.

node2x6x2a:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| thread | sor\_pthread | | | | | | |
| computation | Sync | Total | compute  Speedup | overall  Speedup | Sync/total % | Sync  ratio |
| 1 | 192.92 | 0.013 | 192.94 | 1 | 1 | 0.0067% | 1 |
| 2 | 97.38 | 0.578 | 97.96 | 1.98 | 1.97 | 0.6% | 44.46 |
| 3 | 67.68 | 1.058 | 68.74 | 2.85 | 2.80 | 1.54% | 81.38 |
| 4 | 50.83 | 1.512 | 52.35 | 3.80 | 3.68 | 2.90% | 116.31 |
| 5 | 43.20 | 1.99 | 45.20 | 4.47 | 4.27 | 4.4% | 153.07 |
| 6 | 36.25 | 2.38 | 38.63 | 5.32 | 4.99 | 6.16% | 183.07 |



**Summary:**

The speedup for computation only didn’t exactly follow ideal line because loop overhead and timer function overhead didn’t take into consideration. Since the synchronization overhead cost is growing linearly with the number of threads, the overall speedup slows down with increasing number of threads. Also we can see the percentage: Sync time/ total % raises from 0.6% in 2-thread program to 6% in 6-thread program. So synchronization cost is relatively huge if we have a lot of threads.

## 4 Measure red-black SOR DPJ-based java program

Following the instruction of deterministic parallel java programming, we decide to :

1 mark our data structure with different region

2 write effect summary for each methods. (Illegal and conflict access to memory location will be reported according to the effect summary during the compile-time)

3 write foreach{} DPJ fashion parallel loop instead of java thread.

**Some details:**

region RED,BLACK,COUNTER,GREEN,TIMER;

int max\_it in GREEN;

private int M in GREEN;

private int N in GREEN;

private double W in GREEN;

private double EPS in GREEN;

I assign these attributes to GREEN region because they won’t be written and we are free to read within any parallel loop.

private double[]<RED:[i]>#i[]<RED:[i]:[j]>#j matrix\_red;

private double[]<BLACK:[i]>#i[]<BLACK:[i]:[j]>#j matrix\_black;

Here each double[][] will be mark into different regions to provide isolation.

All elements in matrix\_red are in top level region: RED:\*

All elements in matrix\_black are in top level region: BLACK:\*

Each first dimension element is in different region. For instance:

matrix\_red[1][] is in region RED:[1] and matrix\_red[5][] is in region RED:[5]

(“[5] or [1]” is region name)

Each second dimension element is in different region under first dimension’s region.

For instance: matrix\_red[2][1] is in region RED:[2]:[1]

matrix\_red[1][1] is in region RED:[1]:[1] ; matrix\_red[1][2] is in region RED:[1]:[2]

these two belongs to same sub-region: RED:[1]

Corresponding method will be modified to:

void sor\_red() reads Root,GREEN:\*,BLACK:\* writes RED:\*,TIMER:\*,COUNTER {}

#we read 4 surrouding black nodes and add them to red nodes

void sor\_black() reads Root,GREEN:\*,RED:\* writes BLACK:\*,TIMER:\*,COUNTER {}

#we read 4 surrouding red nodes and add them to black nodes

Wrong effect summary will result in compile-time error.

**Setting1:**

Solving a 1002x4002 array by using Red-Black SOR. (M=1000, N=2000)

omega = 1 Fix iteration = 5000 node2x12x1a

DPJ also provides following options to optimize performance:

--dpj-foreach-split n: Set the branching factor used to split a foreach loop to n. The loop is recursively split into this many branches in each iteration, until the cutoff is reached (see below). The default is 2.

--dpj-foreach-cutoff n: Set the minimum number of foreach iterations allocated to a single task to n. Beyond this point, no more parallel splitting of a foreach loop occurs. The default is 128.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| thread | SOR\_DeNovo | | | | | SOR\_thread | | | | |
| Total | Compute | Sync | Sync% | Speedup | Total | Compute | Sync | Sync% | Speedup |
| 1 | 148.05 | 145.08 | 2.97 | 2% | 1 | 187.47 | 187.13 | 0.34 | 0.2% | 0.79 |
| 2 | 94.16 | 90.03 | 4.13 | 4.38% | 1.57 | 108.37 | 104.93 | 3.4 | 3.17% | 1.4 |
| 4 | 73.91 | 63.67 | 10.24 | 13.85% | 2 | 61.91 | 57.12 | 4.79 | 7.73 | 2.4 |
| 6 | 77.12 | 55.26 | 21.86 | 28.3% | 1.92 | 51.22 | 44.52 | 6.7 | 13.1% | 2.89 |
| 8 | 76.23 | 50.51 | 25.72 | 33.7% | 1.94 | 44.36 | 37.87 | 6.49 | 14.6% | 3.3 |
| 12 | 77.01 | 32.95 | 44.1 | 57% | 1.92 | 49.24 | 41.15 | 8.09 | 16.4% | 3 |

--dpj-num-threads n: Set the number of worker threads used to run the program to n. The default is the number of available processors given to the java virtual machine.

**Setting2:**

Solving a 1002x8002 array by using Red-Black SOR. (M=1000, N=4000)

omega = 1 Fix iteration = 5000 node2x12x1a

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| thread | SOR\_DeNovo | | | | | SOR\_thread | | | | |
| Total | Compute | Sync | Sync% | Speedup | Total | Compute | Sync | Sync% | Speedup |
| 1 | 304.3 | 301.2 | 3.1 | 1% | 1 | 393.3 | 393.0 | 0.3 | 0.1% | 0.77 |
| 2 | 181.59 | 174.7 | 6.85 | 3.77% | 1.67 | 198 | 196.31 | 1.78 | 0.9% | 1.53 |
| 4 | 137.78 | 122.86 | 14.92 | 10.8% | 2.2 | 102.27 | 98.54 | 3.73 | 3.65% | 2.98 |
| 6 | 138.3 | 100.29 | 38 | 27.1% | 2.2 | 86.74 | 76.68 | 10.01 | 11.5% | 3.5 |
| 8 | 140 | 94.03 | 46.01 | 32.8% | 2.2 | 62.02 | 55.73 | 6.29 | 10.1% | 4.9 |
| 12 | 140.9 | 62.02 | 78.8 | 56.3% | 2.16 | 72.03 | 61.39 | 10.63 | 14.7% | 4.2 |

**Summary:**

I expect DPJ has similar performance as Java but the result is disappointing. From the setting1 , DPJ has higher speedup than conventional JAVA when number of threads is smaller than 3. From 2nd figure in Setting1, both programs have almost the same synchronization costs. Starting from 3, DPJ’s synchronization overhead grows dramatically. Even though both computation time is decreasing with the increasing number of working threads. The total time of DPJ program doesn’t decrease. First I think it might be the problem that foreach(I in 1, M){ for(j in 1,N) } the M outer loop in fork-in parallelism costs too much. Therefore in setting2 I double the size of inner loop iteration number N, but the synchronization cost of DPJ program is still much higher than java’s. The reason might be the runtime fork-join technology they provide has low efficiency. In my red-black SOR, I replace :

--barrier

--write to red node

--barrier

--write to black node

By

foreach{ write to red node}

foreach{ write to black node}

the barrier will be implemented by the implicit synchronization point at the end of each foreach loop.

Therefore more fork-join will be called during the runtime leading to huge overhead.

However, Song’s Gaussian Elimination DPJ program shows positive results.

## 5 DPJ applied on Gauss Elimination

The characteristics of DeNovo shown above also give us a possible way to implement Gauss elimination. In each iteration, the procedure of elimination is in such an iteration that is independently conducted on exclusive memory block for each term. Thus, there’s a direct way to just change for loop to foreach loop. Also, it’s needed to put the matrix in region.

What’s more, it’s necessary to make sure noninterference among parallel phase. In fact, the compiler can perform the following noninterference check for indexed foreach loops:

1. Infer the effect set of body.

2. Create a copy of the effect set generated in (1), but replace every occurrence of index-var with a fresh variable that is known to be unequal to index-var. This simulates the effects generated by two distinct iterations of the loop, for which index-var will have distinct values.

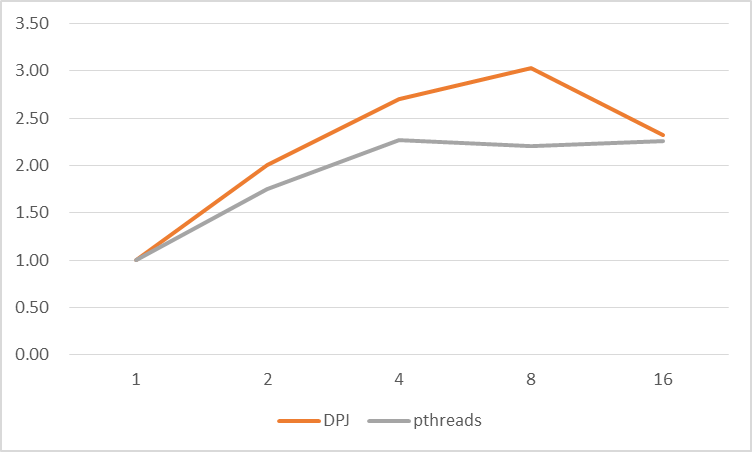
3. Check whether the effect sets generated in (1) and (2) are noninterfering.

If interference is detected, the compiler issues a warning. The DPJ runtime divides the foreach iterations into parallel work according to the programmer-specified granularity.

Noting that the check is just to make sure noninterference rather than confirming interference. I.e. for some cases the compiler will indicate a “warning” while in fact there won’t be problems during running.

Following table shows the comparison between DPJ and pthreads. Testing is conducted on node2x6x2a. And the time recording is only the time for elimination part.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | DPJ | | pthreads | |
| Threads | Time(ms) | Speedup | Time(ms) | Speedup |
| 1 | 5770 | 1.00 | 4590 | 1.00 |
| 2 | 2882 | 2.00 | 2623 | 1.75 |
| 4 | 2132 | 2.71 | 2022 | 2.27 |
| 8 | 1901 | 3.04 | 2079 | 2.21 |
| 16 | 2488 | 2.32 | 2030 | 2.26 |



DPJ provides not only better time consuming performance but also better speedup. Specifically, foreach mechanism enables efficient software controlled coherence mechanisms and powerful communication management, discussed in the above sections.

Thus, it’s expected to have a commensurate performance towards pthreads.

## 6 DeNovo

With DPJ’s help, now we can write data-race-free program. Therefore a read should simply return the value of the last write that either from the reader’s own task or from a task in a previous parallel phase. Directory doesn’t need to remember every sharer at any time because within one phase there will be only one writer. The coherence protocol can be simplified to :

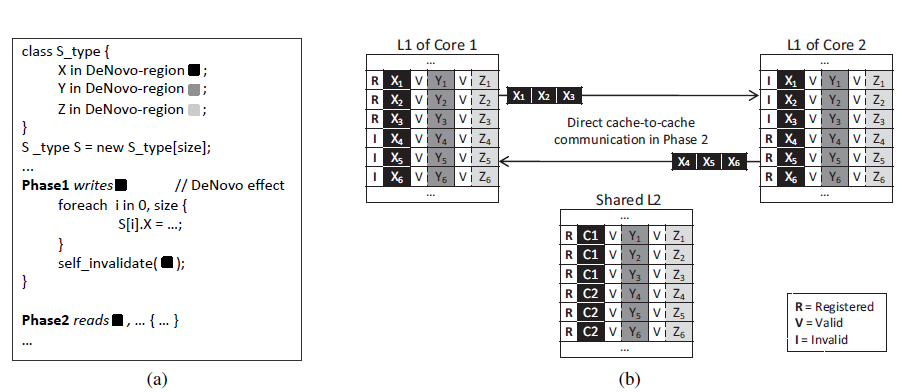


Figure1.Writing and reading from some address will be divided into two phases. At the beginning of phase1, X’s tracing bit in shared L2 will be registered changing from Valid state (Data in L2 is up-to-date) to Registered state (Data in one of L1 is up-to-date). At the end of phase1, self-invalidation will be applied until cache line is “touched” in that particular L1. In phase2, according to the information in L2, if L1 of core2 need the data X1, it will make direct cache-to-cache communication with L1 of core1. Data-race-free guarantee that L1 of core1 in phase1 can freely write X1 and read this dirty X1 without notifying anyone else.

A bit thinking: if we still have to read x in phase3, L1 of core2 will visit L2 and “R”’s state will lead core2 to make cache-to-cache communication with L1 again. But if we let L1 of core2 keep that copy in “volatile” state and runtime predicts next phase’s access pattern. Then tell the L1 of core2 to self-invalidate volatile copy before beginning of next phase.

## Reference

[1] Adve, Sarita V., et al. Comparison of hardware and software cache coherence schemes. Vol. 19. No. 3. ACM, 1991.

[2]Choi, Byn, et al. "DeNovo: Rethinking the memory hierarchy for disciplined parallelism." Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on. IEEE, 2011.

[3] Kelm, John H., et al. "A task-centric memory model for scalable accelerator architectures." Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on. IEEE, 2009.

[4]Kelm, John H., et al. "Rigel: an architecture and scalable programming interface for a 1000-core accelerator." ACM SIGARCH Computer Architecture News. Vol. 37. No. 3. ACM, 2009.