CHALMERS
UNIVERSITY OF TECHNOLOGY

Re-examination August 2020
EDA283/EDA284/DIT360/DIT361
Parallel Computer Architecture

Contact: Miquel Pericàs, Examiner
Email: miquelp@chalmers.se

Written Component

Solutions to Problems

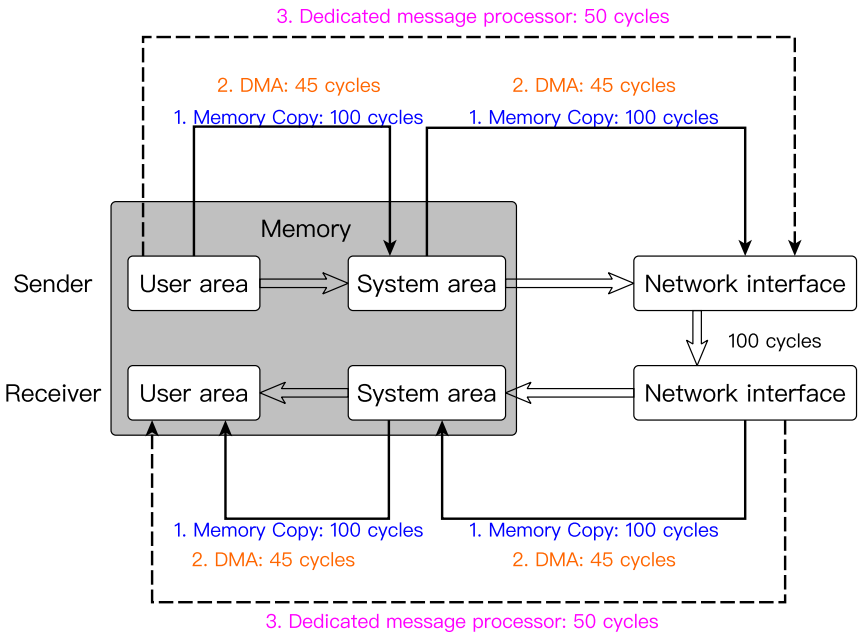
Problem 1 (10p)

Assume the following message passing code, expressed in two versions:
Synchronous version:

CODE FOR THREAD 0:	**CODE FOR THREAD 1:**
B[1] = 1024;	A[1] = 256;
SEND(&B[1], sizeof(float), T1, SEND_B1);	RECV(&A[0], sizeof(float), T0, SEND_B1);
RECV(&B[0], sizeof(float), T1, SEND_A1);	SEND(&A[1], sizeof(float), T0, SEND_A1);
Unrelated computation;	Unrelated computation;

Asynchronous version:

CODE FOR THREAD 0:	**CODE FOR THREAD 1:**
B[1] = 1024;	A[1] = 256;
ASEND(&B[1], sizeof(float), T1, SEND_B1);	ASEND(&A[1], sizeof(float), T0, SEND_A1);
Unrelated computation;	Unrelated computation;
ARECV(&B[0], sizeof(float), T1, SEND_A1);	ARECV(&A[0], sizeof(float), T0, SEND_B1);



Assume that the unrelated computation takes 200 cycles. Consider the following four scenarios:

(a) No special hardware support: It requires five steps of memory copy for a

message from a sender to a receiver. How long does it take to execute the program with synchronous and asynchronous primitives, respectively?

(b) DMA programmed by O/S without support from user messages: But programming DMA costs 15 cycles overhead per each step. What will be the new execution time of synchronous and asynchronous version respectively?

(c) User level messages with O/S support and DMA: Programming DMA costs 15 cycles overhead per step. In this scenario, the message could be delivered directly to the user area from the network interface. However, incoming messages are taken care of by users instead of OS, which costs context switches 100 cycles. What will be the new execution time of synchronous and asynchronous version respectively?

(d) User level messages with a dedicated message processor, shown as dotted line in figure: This dedicated message processor is in the NIC hardware. Hence, it comes for free but still requires a context switch of 100 cycles. What will be the new execution time of synchronous and asynchronous versions, respectively?

Answer to Problem 1:

(a) Synchronous: $100 \times 5 + 100 \times 5 + 200 = 1200$ cycles.
Asynchronous: 500 cycles.

(b) Synchronous: $(45 \times 4 + 100) + (45 \times 4 + 100) + 200 + 15 \times 4 = 820$ cycles.
Asynchronous: $(45 + 15) \times 4 + 100 = 340$ cycles.

(c) Synchronous: $100 + 45 \times 2 + 100 + 100 + 45 \times 2 + 100 + 15 \times 2 + 200 = 810$ cycles.
Asynchronous: $100 + (45 + 15) \times 2 + 100 = 320$ cycles.

(d) Synchronous: $100 + 50 \times 2 + 100 + 100 + 50 \times 2 + 100 + 200 = 800$ cycles.
Asynchronous: $100 + 50 \times 2 + 100 = 300$ cycles.

Problem 2 (9p)

Given a bus-based system implemented as a Chip Muliprocessor (CMP) that has 3 processors (namely P1,P2,P3) where each has a private L1 direct-mapped cache connected to system wide DRAM memory controller, consider the following sequence of memory operations on two 32-bit integers A and B.

```
1. P2: read  A
2. P2: write A, 3
3. P3: write B, 7
4. P3: read  A
5. P3: write A, 2
6. P2: read  A
7. P1: read  B
8. P1: write B, 12
```

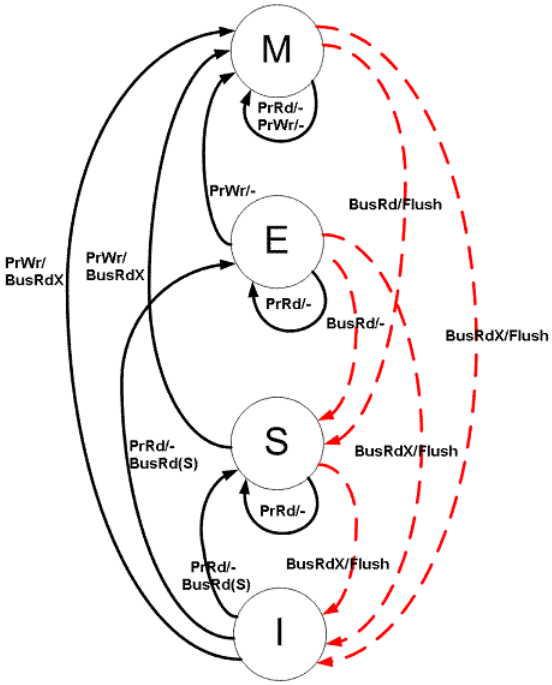


Figure 1: State diagram for MESI protocol Red: Bus initiated transaction. Black: Processor initiated transactions

Assume A and B are initialized to 6, caches are initially empty in invalid state and the cache line size is 64 bytes. Consider also that **A and B are located on the same cache line**. Using the invalidate-based MESI cache coherence protocol with the state diagram given above (showing both bus and processor requests transitions), determine:

(a) The cache line coherence state in each processor.

(b) The contents of A and B in the cache and the memory.

(c) The processor action (`PrRd`, `PrWr`).

(d) The bus action(s) (`BusRd`, `BusUpgr`,`BusRdX`, `flush`).

(e) Design a sequence of 4-5 instructions that result in MESI causing more bus traffic than MSI.

Note: use the tables below to answer parts a - d

Inst	Memory content
-	A(6),B(6)
1	
2	
3	
4	
5	
6	
7	
8	

Inst	P1			
	Cache content	State	PrReq	BusReq
-	-	I	-	-
1				
2				
3				
4				
5				
6				
7				
8				

Inst	P2			
	Cache content	State	PrReq	BusReq
-	-	I	-	-
1				
2				
3				
4				
5				
6				
7				
8				

Inst	P3			
	Cache content	State	PrReq	BusReq
-	-	I	-	-
1				
2				
3				
4				
5				
6				
7				
8				

Inst	Memory content
-	A(6),B(6)
1	A(6),B(6)
2	A(6),B(6)
3	A(3),B(6)
4	A(3),B(6)
5	A(3),B(6)
6	A(2),B(7)
7	A(2),B(7)
8	A(2),B(7)

Inst	P1			
	$	State	PrReq	BusReq
-	-	I	-	-
1	-	I	-	-
2	-	I	-	-
3	-	I	-	-
4	-	I	-	-
5	-	I	-	-
6	-	I	-	-
7	A(2)B(7)	S	PrRd	BusRd
8	A(2)B(12)	M	PrWr	BusRdx

Inst	P2			
	$	State	PrReq	BusReq
-	-	I	-	-
1	A(6)B(6)	E	PrRd	BusRd
2	A(3)B(6)	M	PrWr	-
3	A(3)B(6)	I	-	Flush
4	A(3)B(6)	I	-	-
5	A(3)B(6)	I	-	-
6	A(2)B(7)	S	PrRd	BusRd
7	A(2)B(7)	S	-	-
8	A(2)B(7)	I	-	Flush

Inst	P3			
	$	State	PrReq	BusReq
-	-	I	-	-
1	-	I	-	-
2	-	I	-	-
3	A(3)B(7)	M	PrWr	BusRdx
4	A(3)B(7)	M	PrRd	-
5	A(2)B(7)	M	PrWr	-
6	A(2)B(7)	S	-	Flush
7	A(2)B(7)	S	-	-
8	A(2)B(7)	I	-	Flush

(a-d) Answered in the tables above

(e)
```
1. P2: read  A
2. P1: write A, 3
// events that invalidate all copies of A,B cache line
3. P3: read B
4. P2: write A, 3
```

The code above causes unnecessary flush when instruction 2 is executed, as P2 transitions the cache line from E to I state. In MSI, this wouldn't be the case as transitioning from S to I will happen without a flush. Same applies to instructions 3 and 4. If all copies are invalidated in between (say due to replacement), then similar scenario could happen. However, this is not a common behavior. Typically, if a cache line is read and updated atomically, a process should read and update the cache line exclusively.

Problem 3 (10p)

We want to compare a shared memory (SM) machine and distributed memory (DM) system. Each system has a total of 8 cores. In the SM system, the 8 cores share the main memory (DRAM). The bandwidth of this DRAM memory is 25GB/s. Each core has a peak performance of 10 GFLOPs/s. The DM system consists of 4 nodes, each with 2 cores that are identical to the cores of the SM system. Each node has a DRAM with the same characteristics as the SM system. The nodes are interconnected by a network that consists of a shared bus and has a total bandwidth of 1GB/s.
The program to be executed is an 8-point 2D stencil. C-like pseudocode for the sequential version of this program is shown below:

```
1  #define SIZEX 1024
2  #define SIZEY 1024
3
4  // shared space on which the program is operating
5  float area1[SIZEX][SIZEY], area2[SIZEX][SIZEY];
6
7  void 2dstencil(float in[SIZEX][SIZEY], float out[SIZEX][SIZEY])
8  {
9    for(i=2; i<SIZEX-2; i++)
10     for(j=2; j<SIZEY-2; j++)
11       out[i][j] += 0.125*(in[i+1][j] + in[i-1][j] +
12                           in[i+2][j] + in[i-2][j] +
13                           in[i][j+1] + in[i][j-1] +
14                           in[i][j+2] + in[i][j-2]);
15 }
16
17 main()
18 {
19   float *in, *out, *tmp;
20   in = area1;
21   out = area2;
22
23   for(timestep=0; timestep < MAX_TIMESTEPS; timestep++){
24     2dstencil(in, out);                  // perform stencil
25     tmp = out;  out = in; in = tmp;      // exchange pointers
26   }
27 }
```

Assume that the caches are such that data reuse is possible only within a timestep, but not across timesteps. Within a timestep, caches behave perfectly: they retain all input data needed to compute the kernel and provide unlimited bandwidth to

the cores. Your tasks are:

(a) Calculate the number of memory accesses and floating point operations in `2Dstencil()`. How many arithmetic operations are performed for each byte of data accessed from memory? Determine the performance of a single core when running the `2Dstencil()` function when all other processors are idle. What is the performance when all cores are active for both the SM and for the DM systems?

(b) To parallelize the application, each core computes an equivalent partition of the output grid in the row-direction (X). For the case of distributed memory, and considering the above sizes of SIZEX and SIZEY, how much data needs to be transferred between the 4 nodes after each timestep?

(c) Construct a formula that determines the execution time, for the two above systems, as a function of

   (a) the number of timesteps (MAX_TIMESTEPS), and

   (b) the number of partitions (PARTITIONS).

   Assume unlimited cores (case of SM, one partition per core) and unlimited nodes (case of DM, one partition per node). The memory system (DRAM) architecture remains unchanged. How does the execution time change as the number of partitions increases?

(d) One of your colleagues suggests adding a GPU to accelerate the shared memory system. The GPU has a memory system that has 16 times the bandwidth of the DRAM in the SM system, but has a limited capacity of 8GB. The peak performance of the GPU is 1 TFLOP/s (single-precision). The GPU is connected to the system via an I/O bus such as PCI express, with much lower bandwidth compared to the GPU memory bandwidth. At what speed (FLOPS) can this GPU execute the `2Dstencil()` function? Given the problem size shown above, will it be beneficial to make use of this GPU?

Answer to Problem 3:

Below we give an outline to the solutions:

(a) 9 floating point operations, 2 new memory access (one input, one output). Data type is float = 4bytes, hence := 9 FLOP / 8 byte. Highest theoretical performance is 25GB/s x 9/8 FLOPS/Byte = 28.125 GFlops. This is more than the peak of a single core, hence the single core performance is 10 GFlops. For full system: SM perf will be 28.125 GFlops (memory bound). For DM it will be 80 GFlops (8 cores, compute bound)

(b) 4 columns of SIZEY-2 elements need to be transferred between two adjacent systems. There are three cases so it will be 3 x 4 x (SIZEY-2) x sizeof(float), where sizeof(float) = 4

(c) Case of SM: memory bound, hence perf is 28.125 GFlops. Execution time is then the total number of ops per iteration / 28.125 GFlops and multiplied by the number of timesteps.

Case of DM: For DM, each system perf is 20 GFlops. Execution time has two parts: (1) iteration time = operations per partition / 20 GFlops, (2) communication time = bytes to be transferred (see problem b), divided by interconnect BW (1 GB/s)

(d) To identify whether it is worth to use GPU one must compute the time it takes to execute the kernel on the GPU considering mem BW 400 GB/s and max perf 1 TF. max perf is 400 GB/s x 9 FLOP / 8 byte = 450 GFLOP, hence problem is memory bound. The data can be kept on the GPU for the whole computation since the host side (CPU) does not perform any updates in between. Hence for the above parameters we should use the GPU for the computation.

Problem 4 (8p)

The following C++-like pseudocode represents an application segment that is running on a set of threads in a multicore system. The goal of the segment is to accumulate SIZE values stored in the array `acc_accum[][]`. In the proposed parallelization, the array is already pre-partitioned across NTHREADS threads.

```
1  #define SIZE 1000000
2  #define NTHREADS 4
3
4  std::atomic<int> counter = 0;
5  std::atomic<int> acc_array[NTHREADS][SIZE/NTHREADS];
6  // per-thread array initialized to some values
7
8  // NTHREADS threads all execute the following function in parallel
9  void thread()
10 {
11 int tid = get_thread_number(); // obtain an index to current thread
12 std::atomic<int> inc;
13
14 for(inc = 0; inc < SIZE/NTHREADS; inc++)
15 {
16   counter = counter + acc_array[tid][inc];  // accummulate
17 }
18
19 barrier(); // wait until all threads reach this point
20
21 if (tid == 0) printf ("The final value is %d\n", counter);
22 }
```

The tasks to be completed are as follows:

(a) Discuss the quality of the proposed parallelization. How will it scale if hundreds of of threads are executing?

(b) Propose three optimizations to the code that will make it execute faster.

(c) What is the function of `std::atomic`? How does it impact the processor's pipeline?

(d) Your optimized code is likely to require the use of hardware supported atomic instructions. Atomic instructions may imply also sequential consistency, or they may enable more relaxed memory orders. In the case of your code, discuss what memory orders will be required by the atomic instructions.

Answer to Problem 4:

(a) It want scale because all threads are continuously updating same variable (i.e. lot of invalidations), plus too many variables are marked as atomic which means that parallelism in CPU pipeline is disabled.

(b) (1) privatize counter to a per thread counter, accumulate at the end. (2) increment does not need to be atomic. (3) shared data does not need to be atomic.

(c) atomic makes sure that no intermediate values generated during the computation of an expression can be observed to any other threads. At the processor level, this will block the cache line holding the value and perform the operation on the cache line while disallowing accesses from other threads. Furthermore, by default, the processor pipeline will be drained to enforce sequential consistency

(d) counter needs to have sequential consistency to ensure that updates are performed on the most recent value produced by other threads.

Problem 5 (8p)

In the design exploration of a new chip-multiprocessor (CMP) system that is designed to have 256 cores, the architecture team decides to use a clustered (hierarchical) organization in which each cluster consists of 16 cores, each core has its own private L1 cache and the 16 cores share a L2 cache. Inside of each cluster, the architecture team decides to use bi-directional ring as the on-chip interconnection networks.

(a) Assume a directory-based (DB) protocol is used to maintain cache coherence between cores inside of the same cluster. A core generates a write invalidation request. The latency to communicate between two cores connected directly by a link is 1 cycle and the directory lookup time is 10 cycles. What is the worst latency for the coherence message to reach all possible destinations?

In addition, the CMP system allows cluster-based frequency scaling, which means that the user can power off some clusters in order to allow the rest to operate at a higher frequency.
With the other clusters powered off, users can activate different number of clusters to operate at the following latencies (nanosecond per clock cycle):

When only 1 cluster is active, it operates at 0.4 ns/cycle;
When 2 clusters are active, they operate at 0.7 ns/cycle;
When 4 clusters are active, they operate at 1.3 ns/cycle;
When 8 clusters are active, they operate at 2.5 ns/cycle;
When 16 clusters are active, they operate at 4.2 ns/cycle.

Consider a parallel application that features 500 instructions in the serial part and 25600 instructions in the parallel part. Parallelization also adds extra overhead for creating threads, which adds 5 instructions in the serial part for each created thread.

(b) Assume that running the serial part only needs one cluster to be active. What will be the execution time of the application on this CMP system when using 1, 2, 4, 8 or 16 clusters for the parallel part? Which configuration is the fastest?

(c) Assume now that the system is redesigned such that 8 clusters (0-7) operate at 1.2 ns/cycle and the other 8 clusters (8-15) operate at 0.4 ns/cycle. All clusters are active at the same time on this new CMP. The serial part can be executed on one of the cores of the high frequency clusters (8-15) or low frequency clusters (0-7). How would you manage the execution of the parallel program in order to minimize the execution time?

Answer to Problem 5:

(a) Worst latency = 10 + (16 / 2) * 1 = 18 cycles

(b) 1 cluster:
Serial Part: (500 + 5 $\times$ 16) $\times$ 0.4 ns = 232 ns
Parallel Part: 25600 / 16 $\times$ 0.4 ns = 640 ns
Total: 872 ns

2 clusters:
Serial Part: (500 + 5 $\times$ 32) $\times$ 0.4 ns = 264 ns
Parallel Part: 25600 / 32 $\times$ 0.7 ns = 560 ns
Total: 824 ns

4 clusters:
Serial Part: (500 + 5 $\times$ 64) $\times$ 0.4 ns = 328 ns
Parallel Part: 25600 / 64 $\times$ 1.3 ns = 520 ns
Total: 848 ns

8 clusters:
Serial Part: (500 + 5 $\times$ 128) $\times$ 0.4 ns = 456 ns
Parallel Part: 25600 / 128 $\times$ 2.5 ns = 500 ns
Total: 956 ns

16 clusters:
Serial Part: (500 + 5 $\times$ 256) $\times$ 0.4 ns = 712 ns
Parallel Part: 25600 / 256 $\times$ 4.2 ns = 420 ns
Total: 1132 ns

The Best configuration is 2 clusters.

(c) Serial part executes at high frequency core:
Execution time (serial) = (500 + 5 $\times$ 128) $\times$ 0.4 ns = 456 ns

Since the capacity ratio of high frequency clusters : low frequency clusters = 3 : 1, the best way to minimize the execution time is to keep load balancing between clusters. Then 6400 instructions execute on low frequency clusters, while 19200 instructions execute on high frequency clusters in parallel.
Execution time (parallel) = 6400 / 128 $\times$ 1.2 ns = 60 ns.
Total execution time = 456 + 60 = 516 ns.