Assignment 2 - Solutions

ECSE 420 – Parallel Computing – Fall 2014

November 15, 2014

Q1. **Cache memory & bus-based shared memory multiprocessor** (20%)

Consider a bus-based shared memory multiprocessor system with write-through caches. It is constructed using processors that execute 10^5 instructions/s, and a bus with a peak bandwidth of 10^4 fetches/s. The caches are designed to support a hit rate of 90%. What is the maximum number of processors that can be supported by this system?

We have a hit rate of 90%, we therefore have 10% misses. For the given number of instructions/s, this means $10\% \times 10^5 = 10^4$ memory accesses/s. Thus only 1 processor can be supported.

Q2. **Illinois Protocol** (20%)

Consider a bus-based shared memory multiprocessor system with 2 Processors, Illinois Protocol and write-back caches (initially empty). Both the processors access the shared variables B and C. For the following sequence of commands, write the state of all caches after each executed instruction.

Instruction	P0 cache - VarB	P0 cache - VarC	P1 cache - VarB	P1 cache - VarC
(initially)	I	I	I	I
P0 reads B	E	I	I	I
P1 reads C	E	I	I	E
P0 reads C	E	S	I	S
P1 writes C	E	I	I	M
P0 writes B	M	I	I	M

Table 1: States of caches per executed instruction

Q3. **Test & Set Lock** (20%)

Using the LL and SC instructions, implement the `test & set` lock operation.

The functionality we want to implement is expressed by the following pseudo-code:

```
...
while(t&s (Lock) == 1);
  ... //Critical section
Lock = 0;
...
```

The asked implemention is the following one, where `lock` is the location of the lock stored in memory:

```
t&s:
ADDI R2, R0, #1
LL R1, lock
// Verify if the lock was free,
// (0 means free, 1 means taken).
// If not, loop
BNEZ R1, t&s
SC R2, location
// Verify if the store conditional
// succeeded. If not, loop
BEQZ R2, t&s
// ...Critical Section(C)
unlock: SW lock, #0
```

Q4. **Dragon Protocol** (20%)

Consider a shared memory multiprocessor system with 3 Processors that they all access the same memory location X. All caches are write-back caches, initially empty. The bus operation BusRd requires 9 bytes of bus transfer, while BusUpd requires 6 bytes. Inspect the performance of the Dragon protocol. Write the state of all caches after each executed instruction and the respective bus transaction. Based on the given sequence of actions, give the total bytes transferred using the Dragon protocol.

Total bytes transferred using the Dragon protocol: 51 bytes

Instruction	P1 cache	P2 cache	P3 cache	Bus Transaction
(initially)	–	–	–	–
P1 reads X	E	I	I	BusRd(S')
P1 writes X	M	I	I	–
P1 reads X	M	I	I	–
P1 writes X	M	I	I	–
P2 reads X	Sm	Sc	I	BusRd(S)
P2 writes X	Sc	Sm	I	BusUpd(S)
P2 reads X	Sc	Sm	I	–
P2 writes X	Sc	Sm	I	BusUpd(S)
P3 reads X	Sc	Sm	Sc	BusRd(S)
P3 writes X	Sc	Sc	Sm	BusUpd(S)
P3 reads X	Sc	Sc	Sm	–
P3 writes X	Sc	Sc	Sm	BusUpd(S)

Table 2: States of caches per executed instruction

Q5. **MPI - Function Parallelism** (20%)

Suppose we have a machine with 3 processors. The operation shown below should be executed :

```
...
 D = A*B - A/C;
...
```

One specific task is performed per processor (function parallelism). More specifically:

- P0 computes D - Variables A and D are located only in P0's local memory
- P1 performs multiplication - Variable B is located only in P1's local memory
- P2 performs division - Variable C is located only in P2's local memory

Develop the proper MPI program that implements the aforementioned parallelization scheme. The goal of the current question is to orchestrate the communication scheme, by using MPI Send-Receive pairs properly. Thus, you do not have to submit an individual code file or to compile and execute your code. Directly attach your solution to the submitted PDF file.

Listing 1: `mpi.c` code

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char **argv)
{
  if(argc < 4){
    printf("Usage : ./exec -valueA -valueB -valueC\n");
    return -1;
  }

  int nproc, rank;
  int A, B, C, D;
  int multAB, divAC;

  MPI_Init(&argc, &argv);
  MPI_Status status;

  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if(rank == 0)
  {
    A = atoi(argv[1]);
    MPI_Send(&A, 1, MPI_INT, 1, 10, MPI_COMM_WORLD);
    MPI_Send(&A, 1, MPI_INT, 2, 20, MPI_COMM_WORLD);

    MPI_Recv(&multAB, 1, MPI_INT, 1, 30, MPI_COMM_WORLD, &status);
    MPI_Recv(&divAC, 1, MPI_INT, 2, 40, MPI_COMM_WORLD, &status);

    D = multAB - divAC;
    printf("D=%d\n", D);
  }
  else if (rank == 1)
  {
    B = atoi(argv[2]);
    MPI_Recv(&A, 1 ,MPI_INT, 0, 10, MPI_COMM_WORLD, &status);

    multAB = A*B;
    MPI_Send(&multAB, 1, MPI_INT, 0, 30, MPI_COMM_WORLD);
  }
  else if (rank == 2)
  {
    C = atoi(argv[3]);
    MPI_Recv(&A, 1, MPI_INT, 0, 20, MPI_COMM_WORLD, &status);

    divAC = A/C;
    MPI_Send(&divAC, 1, MPI_INT, 0, 40, MPI_COMM_WORLD);
  }

  MPI_Finalize();
  return 0;
}
```