PAR – Final Exam Theory/Problems– Course 2023/24-Q1

January 17th, 2024

Problem 1 (6.0 points)

Given the following code fragment written in C:

```
int m[N][N];
...
for (int i=0; i<N; i++) {
   for (int k=0; k<N; k++) {
       m[i][k] = calculation (m[N-1-i][k], m[i][k]); // 100 time units
   }
 }
```

We ask you to:

1. (1.0 Points) Assume 1) a very fine-grain strategy in which a task is defined for each single k loop iteration, with cost equal to 100 t.u. (time units); 2) function `calculation` only reads the two values received as arguments and does not modify any other memory locations; and 3) the rest of variables in the program are stored in registers of the processors. **Also assume $N = 4$ for the following two subquestions:**

   (a) Draw the *Task Dependency Graph (TDG)*, identifying each task in the *TDG* with a label (i, k), being i and k the values of the `i` and `k` loop control variables, respectively.

   (b) Compute the values for the metrics T_1, T_∞ and P_{min}.

2. (2.5 Points) Write an *OpenMP* parallel version of the code following the most appropriate *geometric data decomposition strategy* for matrix `m` considering that your parallel code should: a) minimize the synchronization overhead among implicit tasks based on the previous *TDG* analysis; and b) guarantee that the load unbalance is limited to N elements (i.e. the number of elements in a row or column of the matrix). Assume that matrix `m` is allocated in memory by rows and that N is a multiple of 2 and not necessarily multiple of the number of processors to be used to execute the program in parallel.

3. (2.5 Points) Modify the previous *OpenMP* parallel version or write a new one and, if necessary, change the definition of matrix `m` to follow the most appropriate *geometric data decomposition strategy* considering that: a) the program is executed on a parallel machine where memory lines are 128 bytes long; b) matrix `m` is allocated in memory so that its first element is aligned to the start of a memory line and `sizeof(int)` $=$ 4 bytes; and c) you can not consider that N is multiple of the size of one element. Your proposed solution should: a) maximize parallelism among implicit tasks; b) maximize data locality and reduce coherence traffic; and c) minimize load unbalance.

Problem 2 (4.0 points) Assume a NUMA system with two NUMAnodes, each NUMAnode with 16GB of main memory and 2 processors (cores). Each processor has its own local cache memory of 4MB (cache line size of 128 bytes). Data coherence is maintained using *Write-Invalidate MSI* protocols, with a Snoopy attached to each per–core local cache memory to provide coherency within each NUMAnode and directory-based coherence among the two NUMAnodes.

We ask you to:

1. (2.0 Points) Compute the total number of bits that are used by each snoopy to maintain the coherence between caches inside a NUMAnode and, the total number of bits in each node directory to maintain coherence among NUMAnodes.

2. (2.0 Points) Let's consider the following definition of matrix `m` with a given value of `N`:

   ```
   #define N 32
   int m[N][N];
   ```

   Assume that a) matrix `m` is allocated in memory aligned to the start of a memory line and b) `sizeof(int)` = 4 bytes.

   The code initializing matrix `m` is executed entirely by $core_0$ (on $NUMAnode_0$) so at the end of the initialization the entire matrix is loaded into the cache of $core_0$. Consequently, all memory lines and cache lines that hold matrix `m` are in *Modified state* with *presence bits = 01* in the directory of the *home* NUMAnode. **We ask you to:** fill in the attached table the sequence of processor commands (Core), cache Miss or Hit, bus transactions within NUMA nodes indicating clearly which attached Snoopy generated the command (Snoopy), state for each cache line affected (for each cache memory of the system, MC_0 to MC_3), state for each memory line (Directory state) and the presence bits, to keep cache coherence, AFTER the execution of each of the following of commands:

   - $core_0$ from $NUMAnode_0$ reads `m[0][0]`
   - $core_0$ from $NUMAnode_0$ reads `m[1][24]`
   - $core_3$ from $NUMAnode_1$ writes `m[0][24]`
   - $core_3$ from $NUMAnode_1$ reads `m[1][24]`
   - $core_0$ from $NUMAnode_0$ writes `m[0][0]`
   - $core_3$ from $NUMAnode_1$ writes `m[1][24]`

Solution:

Action	Core	Cache Miss/Hit	Snoopy	State MC_0	State MC_1	State MC_2	State MC_3	Directory State	Presence bits
$core_0$ reads m[0][0]									
$core_0$ reads m[1][24]									
$core_3$ writes m[0][24]									
$core_3$ reads m[1][24]									
$core_0$ writes m[0][0]									
$core_3$ writes m[1][24]									