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

January 17^{th}, 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
   }
 }
```

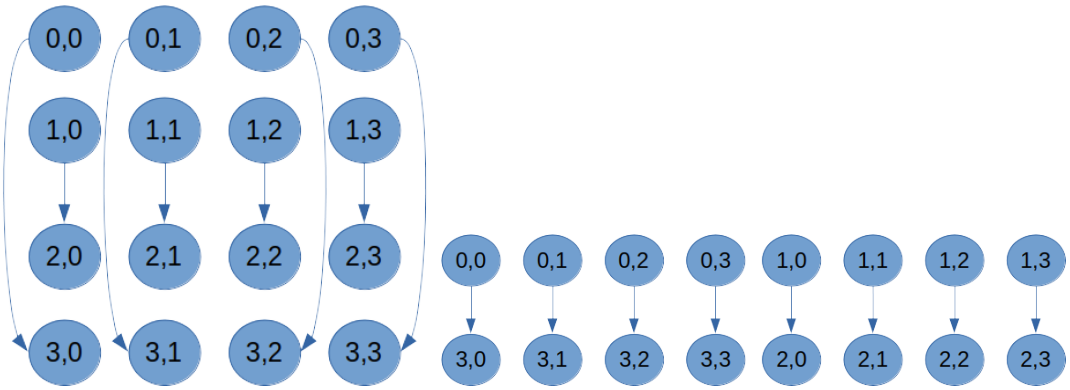
We ask you to:

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.

   Solution:

   Two possible views of the TDG.



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

   Solution:

   $T_1 = 16 \times 100 = 1600$ time units.

   $T_\infty = 2 \times 100 = 200$ time units.

   $P_{min} = 8$.

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.

   Solution:

```
int m[N][N];
...
```

```
#pragma omp parallel
{
 int id = omp_get_thread_num();
 int nths = omp_get_num_threads();
 int BS = N/nths;
 int mod = N%nths;
 int start = id*BS;
 int end = start + BS;
 if (mod > 0) {
    if (id < mod) {
       start += id;
       end = start + BS + 1;
    }
    else {
       start += mod;
       end += mod;
    }
 }
 for (int i=0; i<N; i++) {
   for (int k=start; k<end; k++) {
       m[i][k] = calculation (m[N-1-i][k], m[i][k]);
   }
 }
```

We apply a *block geometric data decomposition by columns*. There are RAW (true) dependencies among elements from the same column, consequently blocks are defined by columns, ensuring no need for synchronization among implicit tasks. The size of the blocks are adjusted as N is not guarantee to be a multiple of the number of threads. In this way, the unbalance is limited to N elements (an entire column).

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.

Solution:

```
#define MEMLINESIZE 128
#define X MEMLINESIZE/sizeof(int)
int m[N][N + (X - N % X)];
...
#pragma omp parallel
{
 int id = omp_get_thread_num();
 int nths = omp_get_num_threads();
 int BS = (N/2)/nths;
 int mod = (N/2)%nths;
 int start = id*BS;
 int end = start + BS;
 if (mod > 0) {
    if (id < mod) {
       start += id;
       end = start + BS + 1;
    }
    else {
```

```
        start += mod;
        end += mod;
     }
  }

  for (int i=start; i<end; i++) {
    for (int k=0; k<N; k++) {
        m[i][k] = calculation (m[N-1-i][k], m[i][k]);
        m[N-1-i][k] = calculation (m[i][k], m[N-1-i][k]);
    }
  }
}
```

In the proposed solution we apply a *block geometric data decomposition strategy by rows* on the first half of the matrix `m`. The rows on the second half are processed by the thread that processes the mirror row in the first half of the matrix, so the code is modified accordingly. In this way dependencies are preserved and there is no need for synchronization between implicit tasks. In order to minimize coherence traffic, we avoid false sharing by adding padding as extra columns at the end of each row of the matrix, so that we complete an entire memory line in case N is not a multiple of the memory line size. In this way we avoid false sharing. Data locality is preserved as the matrix processing is performed by rows. Finally, unbalance is minimized to two rows (2N elements) in the worst case.

Another solution could be to modify the proposed solution of the previous part, and apply a *block cyclic gemotreci data decomposition by columns*:

```
#define MEMLINESIZE 128
#define BS MEMLINESIZE / sizeof(int)
int m[N][N + (BS - (N % BS)];
...
#pragma omp parallel
{
 int id = omp_get_thread_num();
 int nths = omp_get_num_threads();

 for (int i=0; i<N; i++) {
   for (int kk=id*BS; kk<N; kk+=nths*BS) {
      for (int k=kk; k<min(kk+BS,N); k++) {
       m[i][k] = calculation (m[N-1-i][k], m[i][k]);
   }
 }
}
```

Padding is also added to avoid false sharing and consequently minimize coherence traffic. The size of the block ensures data locality. This solution has the disadvantage with respect to the previous proposed one, that the unbalance in the worst case could be up to (BS-1) columns, that is (BS-1)*N elements.

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.

   Solution:

   Intra-node coherence:

   The number of cache lines is calculated dividing the memory in a cache (4MB) by the cache line size (128B). Therefore: Number of cache lines $= \frac{4MB}{128B} = \frac{4\times 2^{20}}{2^7} = 2^{15}$ entries

   We need 2 bits per entry, so the total number of bits used by each snoopy is: 2^{16}

   Inter-node coherence:

   The number of entries in the directory structure per NUMA-node is calculated dividing the memory in a NUMA-node (16GB) by the memoy line size (128B). Therefore:

   Number of Directory Entries $= \frac{16GB}{128B} = 2^{27}$ entries

   Each entry has 4 bits: 2 Presence bits and 2 bits for the Status

   Thus, with 4 bits per directory entry, the total number of bits per NUMAnode is: $4 \times 2^{27} = 2^{29}$

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]									

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]	$PrRd_0$	Hit	-	M	-	-	-	M	01
$core_0$ reads m[1][24]	$PrRd_0$	Hit	-	M	-	-	-	M	01
$core_3$ writes m[0][24]	$PrWr_3$	Miss	$BusRdX_3$ $BusRdX$ in $NumaNode_0$ issued by Hub $Flush_0$	I	-	-	M	M	10
$core_3$ reads m[1][24]	$PrRd_3$	Miss	$BusRd_3$ $BusRd$ in $NumaNode_0$ issued by Hub $Flush_0$	S	-	-	S	S	11
$core_0$ writes m[0][0]	$PrWr_0$	Miss	$BusRdX_0$ $BusRdX$ in $NumaNode_1$ issued by Hub $Flush_3$	M	-	-	I	M	01
$core_3$ writes m[1][24]	$PrWr_3$	Hit	$BusUpgr_3$ $BusUpgr$ in $NumaNode_0$ issued by Hub	I	-	-	M	M	10