PAR – In-Term Exam – Course 2020/21-Q2

April 21st, 2021

Problem 1 (1.5 points) Given the following code:

```
#define N 8

for(i=0;i<N;i++) {
  tareador_start_task("task-b");
  b[i] += foo(i);
  tareador_end_task("task-b");
}

for(i=1;i<N-1;i++) {
  tareador_start_task("task-a");
  a[i] = b[i-1] + b[i+1];
  tareador_end_task("task-a");
}
```

Assume that 1) the cost of executing one instance of the loop body for the two loops is t_c; and 2) the parallel task decomposition strategy is the one indicated with *Tareador* annotations. **We ask you to answer** the following questions:

1. How many tasks of type `task-b` and `task-a` are created?

   Solution:

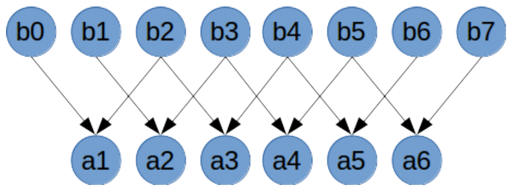
   The first loop iterates from 0 to $N-1$; one task is created per iteration. Therefore, N tasks of type `task-b` are created.

   The second loop iterates from 1 to $N-2$; one task is created per iteration. Therefore, $N-2$ tasks of type `task-a` are created.

2. Draw the TDG of the parallel code, indicating the cost for each task and dependences between tasks.

   Solution:

   Each `task-a` created in iteration i is shown in the TDG with a_i. Each `task-b` created in iteration i is shown in the TDG with b_i. Each task has a cost of t_c.

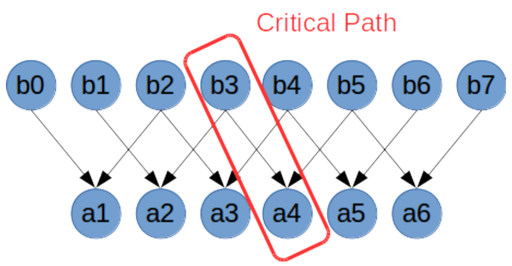


3. Compute T_1, T_∞ and P_{min}.

   Solution:

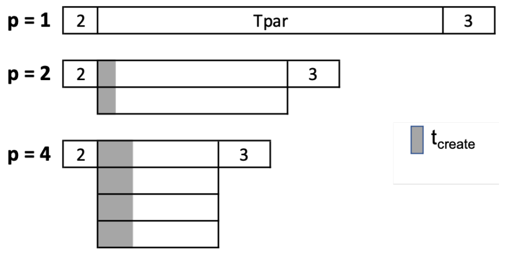
   Given the previous TDG:

   - $T_1 = N \times t_c + (N-2) \times t_c = (2N-2) \times t_c$.
   - $T_\infty = 2t_c$. One `task-b` plus one `task-a` as it is shown in the following figure with one of the possible critical paths.



- $P_{min} = N$. This is because all tasks (N) of type `task-b` should be able to be executed at the same time.

Problem 2 (1.5 points) Consider the following execution timelines that show the *strong scaling* behaviour of a simple parallel program composed of a single parallel region. The first timeline corresponds with the execution on a single processor, showing the sequential ($T_{seq} = 2 + 3$) and parallel (T_{par} unknown) execution bursts that define T_1. The second and third timeline correspond with the execution on two and four processors, respectively, in which we added the overhead of task creation $t_{create}(p) = \alpha \times p$ (which is proportional to the number of processors used in the parallel execution); therefore, $T_p = T_{seq} + \frac{T_{par}}{p} + t_{create}(p)$.



We ask you to:

1. Calculate the value for T_{par} assuming that $\varphi = \frac{4}{5}$.

   Solution: $T_1 = T_{seq} + T_{par}$ and $\varphi = T_{par} \div T_1$. Since $T_{seq} = 5$ and $\varphi = \frac{4}{5}$, one can easily obtain $T_{par} = 20$.

2. Obtain the value for the proportionality constant α assuming that $S_2 = 1.65$.

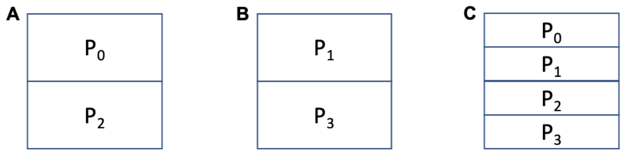
   Solution: $S_p = T_1 \div ((1 - \varphi) \times T_1 + (\varphi \times (T_1/p) + (\alpha \times p)))$. From this expression applied to $p = 2$ one can easily get $\alpha = 0.075$.

3. Obtain the value for S_4.

   Solution: From the same expression, by simply substituting the value for α just obtained one can get $S_4 = 2.42$.

Problem 3 (2 points) Consider a distributed memory architecture in which accessing the data stored in a remote processor through an interconnection network implies a data sharing overhead of $t_s + m \times t_w$ (being t_s the *start-up* time, m the number of elements being accessed, and t_w the *per-element* transfer time). At any time, a processor can simultaneously perform one remote access and serve one remote access, but only one of each kind. The data sharing model assumes that local accesses take zero overhead.

Assume the following simple parallel region to be executed with 4 processors and the distribution of matrices shown in the following figure:



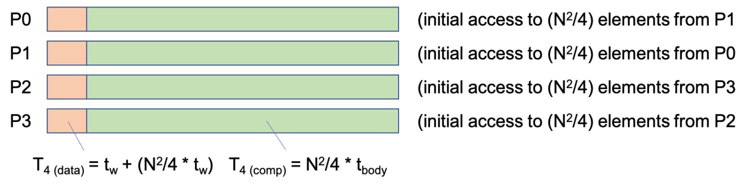
```
#define N 64
#define N_THREADS 4
int A[N][N], B[N][N], C[N][N];
...
#pragma omp parallel num_threads(N_THREADS)
{
    int thid = omp_get_thread_num();
    int chunk = N / N_THREADS;

    for (int i = thid * chunk; i < (thid + 1) * chunk; i++)
        for (int j = 0; j < N; j++)
            C[i][j] = A[i][j] + B[i][j];
}
```

with processor P_x executing the OpenMP thread with identifier x; each processor can start the execution of its thread as soon as it has all the data elements that are needed to execute the thread. The execution time for one iteration of the loop body is t_{body}.

We ask you:

1. Draw a temporal diagram (timeline) showing the execution of the threads by each processor and the data sharing overheads that occur.

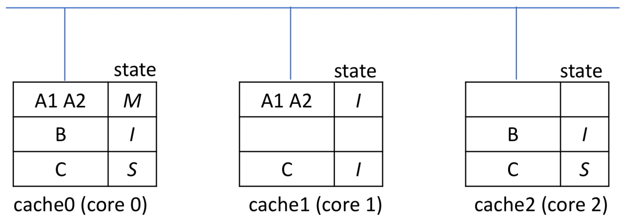
   Solution: Each processors needs to access to half of the elements stored in another processor, incurring in a data sharing overhead. After that, each processor can proceed with the execution of its computation burst.



2. Obtain the expression for T_4 including both computation $T_{4(comp)}$ and data sharing $T_{4(data)}$.

   $T_4 = T_{4(comp)} + T_{4(data)}$
   $T_{4(comp)} = (N^2 \div 4) \times t_c$
   $T_{4(data)} = t_s + (N^2 \div 4) \times t_w$

Problem 4 (2 points) Given an SMP system with 3 processors (cores), each with its own cache memory and a shared main memory. Data coherence in the system is maintained using *Write-Invalidate MSI protocol*, with a Snoopy attached to each cache memory. The initial state of the cache memories is shown in the following representation of the system, in which only three cache lines are represented for each cache memory, with variables: A1, A2, B, C. Observe that variables A1 and A2 reside in the same cache line, while the other two reside in a different cache line each.



Assuming the following sequence of accesses to variables, **we ask you to** fill in the table indicating if each access makes a Hit or Miss in the affected cache, the Bus transactions ($BusRd_k$, $BusRdX_k$, $BusUpgr_k$, $Flush_k$, being k the core that provokes the transaction) that are generated and the state (M, S or I) of only the cache lines that contain the variable in each cache memory after each access. In the observations column indicate who is providing the line when a cache requires it and when main memory is updated.

Memory access	hit/miss	Bus transaction(s)	Cache line state			Observations
			Cache 0	Cache 1	Cache 2	
core 0 reads A1						
core 1 reads A2						
core 2 writes C						
core 2 reads C						
core 0 writes A1						
core 0 reads B						
core 2 reads B						
core 1 writes C						

Solution:

Memory Access	hit/miss	Bus transaction(s)	Cache Line State			Observations
			Cache 0	Cache 1	Cache 2	
core 0 reads A1	hit	–	M			(2)
core 1 reads A2	miss	BusRd/Flush	S	S		(2), (3)
core 2 writes C	hit	BusUpgr	I	I	M	(2)
core 2 reads C	hit	–	I	I	M	(2)
core 0 writes A1	hit	BusUpgr	M	I	I	(2)
core 0 reads B	miss	BusRd	S		I	(1)
core 2 reads B	miss	BusRd	S		S	(1)
core 1 writes C	miss	BusRdX/Flush	I	M	I	(2), (3)

Legend for column observations: (1) Main memory provides the line. (2) Cache provides the line. (3) Main memory is updated.

Problem 5 (1.5 points) Let's assume a multiprocessor system with a hybrid NUMA/UMA architecture composed of 6 identical NUMAnodes, each one with 8 GB (gigabytes, i.e. 2^{30} bytes) of main memory and 3 processors with their own private cache of 128 MB (megabytes, i.e. 2^{20}). Memory and cache lines are 16 bytes wide. Data coherence is maintained using *Write-Invalidate MSI protocol* within each NUMAnode and using a *Write-Invalidate MSU Directory-based* cache coherency protocol among NUMAnodes. **We ask you to** answer the following questions:

1. Compute the total number of bits that are necessary in each cache memory to maintain the coherence between the caches **inside a NUMA node**.

   Solution:

   One needs 2 state bits (MSI) for each line of cache memory. Each cache memory has $(128 \times 2^{20}) \div 16$ lines, that is 2^{23} lines; therefore the number of bits per cache is $2^{23} \times 2 = 2^{24}$ bits, that translated into MB is $(2^{24} \div 8) \div 2^{20} = 1$ MB. Since there are 18 processors, each one with its own private cache, the whole system needs 36 MB to keep cache coherence.

2. Compute the total number of bits that are necessary in each node directory to maintain the coherence **among NUMA nodes**.

   Solution:

   One needs 2 state bits (MSU) and 6 presence bits for each line of main memory. For the overall 48 GB, this is $(48 \times 2^{30}) \div 16$ lines, that is 48×2^{26} lines; therefore the number of bits in the directory is $48 \times 2^{26} \times (6 + 2) = 48 \times 2^{29}$ bits, that translated into MB is $((48 \times 2^{29}) \div 8) \div 2^{20} = 48 \times 2^{6} = 3072$ MB.

Problem 6 (1.5 points) Given the following OpenMP code:

```
#define N         (1<<8)    /*  256 */
#define N_THREADS (1<<6)    /*   64 */

int b[N];
int result=0;

#pragma omp parallel shared(result) num_threads(N_THREADS)
{
  int thid = omp_get_thread_num();
  int chunk = N / N_THREADS;

  for (int i = thid * chunk; i < (thid + 1) * chunk; i++) {
    b[i] += foo(i);
    result += b[i];
  }
}
```

Assume a cache-coherent system with 64 processors with write–invalidate coherence protocol and cache lines of 128 bytes. Also assume that the execution of `foo(i)` does not perform any memory accesses, `int` size is 4 bytes and variable `result` is placed at the first position of a memory line (not sharing any memory line with the elements of vector `b`). **We ask you to answer** the following questions:

1. Briefly describe which part of the code provokes a data race (true sharing) situation.

   Solution:

   All threads are **sharing and updating variable `result` with no synchronization** within the for loop. That provokes a possible data race condition reading and writing to result by all threads in the parallel region.

2. Briefly describe which part of the code provokes a false sharing situation (because of the given values for N and $N\_THREADS$).

   Solution:

   A `chunk` of consecutive iterations of the for loop is done in parallel by each thread, reading and updating consecutive elements of b vector ($b[i]+=...$). Although thoses `chunk`'s do not share iterations neither $b[i]$ elements, threads updating two consecutive `chunk`'s of elements of vector b share the same cache line, provoking a false sharing situation.

   Detail:

   In our case, one cache line is 128 bytes and each b element is an `int` of 4 bytes. Therefore, each cache line can contain $128 bytes \times \frac{1 element}{4 bytes} = 32 elements$. For the N and `N_THREAD` given, each thread processes a `chunk` of iterations equal to $N/$`N_THREADS`. That means that each thread processes $256/64 = 16$ consecutive elements of b of type int (half cache line). Therefore, two consecutive threads share the same cache line (half each) when accessing their `chunk` of elements.

3. For the given value of N, compute the maximum number of threads $N\_THREADS$ that can be used (larger than 1 and smaller than or equal to 64) to avoid the occurrence of the false sharing situation. Reason your answer.

   Solution:

   We need that each cache line, containing b elements, be accessed by only one thread. That means that each thread should process 32 elements (number of elements that a cache line can contain). Then, `chunk` is computed as $N/$`N_THREADS`. Therefore, we only need to solve this equation: $chunk = 32 = 256/N\_THREADS$.

   That means that `N_THREADS` should be 8.