PAR – In-Term Exam – Course 2021/22-Q1

November 8^{th}, 2021

Problem 1 (2 points)

We have a sequential code that we want to parallelize. Our code has four disjoint parts executed one after the other, namely *Start*, *Init*, *Compute* and *End*, which take 1, 10, 100 and 2 time units, respectively.

1. If we only parallelize the *Compute* phase, which would be the value for the parallel fraction ϕ? Assuming that *Compute* can be perfectly parallelized, and there are no overheads resulting from its parallelization, which would be the value for the ideal speed–up $S_{p\to\infty}$?

   Solution:

   $\phi = \frac{100}{113} = 0.885$; $1 - \phi = 0.115$

   Then, according to Amdahl's law when the number of processors $P \to \infty$:

   $S_{p\to\infty} = \frac{1}{1-\phi} = \frac{1}{0.115} = 8.69$

2. Our system, however, has a parallelization overhead proportional to the number of processors being used. Which would be the value for the speed–up when using 10 processors (S_{10}) if the parallelization of *Compute* can be perfectly parallelized but incurs in an overhead of 0.001 time units per processor being used?

   Solution:

   Considering the overheads and Amdahl's law:

   $S_P = \frac{1}{1-\phi+\phi/P+0.001*P/T_1}$

   Thus,

   $S_{10} = \frac{1}{1-\phi+\phi/10+0.001*10/T_1} = 4.91$

   or, alternatively, just applying the definition of $S_p = T_1/T_p$:

   $S_{10} = \frac{T_1}{T_{10}} = \frac{113}{13+100/10+0.001*10} = \frac{113}{23.01} = 4.91$

3. Next, we parallelize the *Init* phase on two processors. Regrettably, in this *Init* phase the parallelization cannot be scaled beyond two processors. Assuming again that there are no overheads due to the parallelization, which would be the new value for S_{10} in this case?

   Solution:

   Since a part of the code cannot be perfectly parallelized we cannot use Amdahl's law here. Thus, our only choice in this case is using the definition $S_P = \frac{T_1}{T_P}$.

   $T_{10} = 1 + 5 + 10 + 2 = 18$

   Therefore: $S_{10} = \frac{T_1}{T_{10}} = \frac{113}{18} = 6.277$

Problem 2 (2.5 points)

Assume a program composed of two parallel regions: *Region A* and *Region B*; both regions scale ideally when executed with P processors. There is no code outside these two regions in the program. The first region, *Region A*, is basically a double nested loop reading matrix `Matrix_a` and writing into matrix `Matrix_b`:

```
for (int row = 0; row < N; row++)
  for (int col = 0; col < N; col++)
        Matrix_b[row][col] = foo(Matrix_a[row][col]);
```

For this region the programmer has implemented a task decomposition in which each task computes N/P consecutive iterations of the `row` loop, with tasks computing blocks of rows assigned to processors in ascending order. Once the execution of *Region A* is finished, the program proceeds with *Region B*, which is also a double nested loop reading matrix `Matrix_b` and writing into matrix `Matrix_c`:

```
for (int col = 0; col < N; col++)
  for (int row = 0; row < N; row++)
         Matrix_c[row][col] = goo(Matrix_b[row][col]);
```

For this region the programmer has implemented a task decomposition in which each task computes N/P consecutive iterations of the `col` loop, again with tasks computing blocks of columns assigned to processors in ascending order. After the execution of *Region B* the program terminates.

The three matrices have N rows and N columns, but are distributed in three different ways: `Matrix_a` is totally stored in the memory of processor 0; `Matrix_b` is distributed by rows among all P processors, so that each processor stores N/P consecutive rows in its memory (blocks of rows mapped to processors in ascending order); and `Matrix_c` is distributed by columns among all P processors, so that each processor stores N/P consecutive columns in its memory (blocks of columns mapped to processors in ascending order).

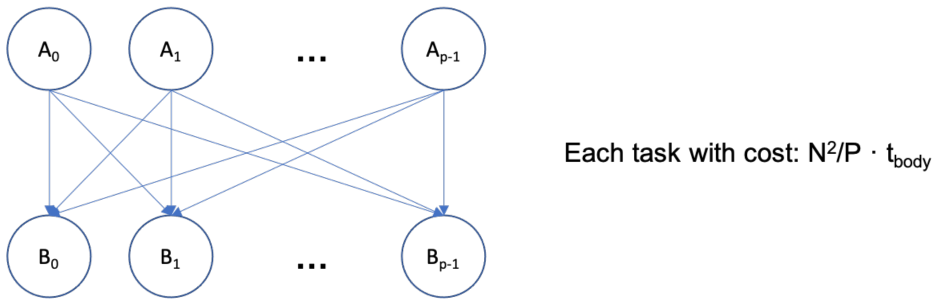
You can assume the data sharing model explained in class in which the overhead to perform a remote access is $t_s + t_w \times m$, being t_s the start-up time, t_w the time to transfer one element and m the number of elements to be transferred. At any time, each processor can simultaneously make one remote access to a different processor and serve one remote access from another processor. You can also assume that the execution of body of each innermost loop takes t_{body}.

We ask you to:

1. Draw the *Task Dependence Graph* (TDG) for the program described above, indicating the cost of each task.

   Solution:

   The TDG includes P nodes, each with a cost of $\frac{N^2}{P} \times t_{body}$, for *Region A* and P nodes, each also with a cost of $\frac{N^2}{P} \times t_{body}$, for *Region B*. Each task in *Region B* depends on all tasks in *Region A*, as shown in the TDG below.

2. Identify which remote accesses have to be performed during the execution of the parallel program, clearly identifying the processors involved in each remote access, the number of elements that need to be transferred and when the remote accesses should occur.

   Solution:

   Before the computation in *Region A* can start, each processor should access to those elements of `Matrix_a` that are needed, that is: each processor from 1 to $P-1$ should access to N/P consecutive rows (each one with N elements) from processor 0. Since this processor can only serve one remote request at a time, all these remote accesses are sequentialised. Before the computation in *Region B* can start, each processor should access to those elements of `Matrix_b` that are needed, that is: each processor (from 0 to $P-1$) should access to N/P consecutive rows (each one with N/P consecutive columns) from all other $P-1$ processors; and alternatively, each processor has to serve $P-1$ requests from the other processors which should be sequentialised.

3. Write the expression that determines the execution time with P processors, T_P, clearly identifying the contribution of the computation time and the overheads caused by data sharing.

Solution:

Remote accesses before starting the parallel execution in *Region A* are sequentialised because processor 0 can only serve 1 remote access at a time: this is why the cost of a remote access ($t_s + t_w \times N^2/P$) is multiplied by P-1. Similarly, the P-1 remote accesses to the same processor before starting the parallel execution in *Region B* are also sequentialised because a processor can only serve 1 remote access at a time: this is why the cost of a remote access ($t_s + t_w \times (N/P)^2$) is multiplied by P-1 too. Therefore, the expression for T_P is:

$T_p = T_P^{comp} + T_P^{data}$

$T_P^{comp} = N^2/P \times t_{body} + N^2/P \times t_{body} = 2 \times N^2/P \times t_{body}$

$T_P^{data} = (P-1) \times (t_s + t_w \times N^2/P) + (P-1) \times (t_s + t_w \times (N/P)^2)$.

Problem 3 (3 points) Assume a multiprocessor system with a hybrid NUMA/UMA architecture. The multiprocessor is composed of 2 identical NUMAnodes, each with 12 Gbytes of main memory. Each NUMAnode has 2 processors, each with its own private cache of 16 Mbytes. Memory and cache lines are 128 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. **First, we ask you to** answer the following two questions:

1. Compute the total number of bits that are necessary **in each cache memory** to maintain the coherence between caches **inside a NUMAnode**. Indicate also the function of those bits.

   Solution:

   We need 2 state bits (MSI) for each line of cache memory. Each cache memory has $(16 \times 2^{20}) \div 128$ lines, that is 2^{17} lines; therefore the number of bits per cache is $2^{17} \times 2 = 2^{18}$ bits.

2. Compute the total number of bits that are necessary **in each NUMAnode's directory** to maintain the coherence **among NUMAnodes**. Indicate also the function of those bits.

   Solution:

   We need 2 state bits (MSU) and 2 presence bits for each line of main memory. For the overall 12 GB, this is $(12 \times 2^{30}) \div 128$ lines, that is 12×2^{23} lines; therefore the number of bits in the directory is $12 \times 2^{23} \times (2+2) = 3 \times 2^{27}$ bits.

Now, given the following declaration for vector `x`:

```
#define N 1024
int x[N];
```

and assuming that: 1) the initial memory address of vector `x` is aligned to the start of a memory/cache line; 2) the size of an `int` data type is 4 bytes; and 3) processors 0 and 1 belong to NUMAnode0 and processors 2 and 3 belong to NUMAnode1. **We ask you to:**

3. Complete the table in the provided answer sheet with the necessary missing information: type of memory access (`read`/`write`), affected cache line (numbered from the first position where vector `x` is allocated), access in cache (hit/miss), CPU command for processor k ($PrRd_k/PrWr_k$), Bus transaction(s) from Snoopy in processor k ($BusRd_k/BusRdX_k/BusUpgr_k/Flush_k/Nothing$), cache line states (`I`/`S`/`M`), NUMA commands (yes/no), directory entry state (`U`/`S`/`M`) and presence bits (0/1, where the lowest ordered bit, the rightmost one, corresponds to NUMAnode0), to keep cache coherence, **AFTER the execution of each** memory access. **Note**: We are not asking for the coherence commands exchanged between NUMAnodes, we are only asking the Bus transactions within NUMAnodes to keep coherence resulted from local or remote memory access to NUMAnodes.

Solution:

Observe that cache line size is 128 bytes and each integer is 4 bytes long. Therefore, 32 consecutive elements of vector x fit in one cache line. In the table below this means that `x[4]`, `x[16]` and `x[20]` belong to the same line (line 0) and `x[32]` is in a different line (line 1).

Memory access	Affected line	Hit/Miss	CPU command	Bus transaction(s)	Cache line state 0	1	2	3	NUMA commands	Directory entry State	Presence bits
Processor 1 writes x[4]	0	Miss	PrWr1	BusRdX1	I	M	I	I	No	M	01
Processor 2 reads x[16]	0	Miss	PrRd2	BusRd2 / Flush1	I	S	S	I	Yes	S	11
Processor 3 writes x[32]	1	Hit	PrWr3	BusUpgr3	I	I	I	M	Yes	M	10
Processor 0 writes x[32]	1	Miss	PrWr0	BusRdX0 / Flush3	M	I	I	I	Yes	M	01
Processor 2 writes x[20]	0	Hit	PrWr2	BusUpgr2	I	I	M	I	Yes	M	10

Problem 4 (2.5 points) Given the following OpenMP code:

```
#define N         (1<<18)   /*  256*1024 */
#define N_THREADS (1<<4)    /*   16 */

int b[N];
int a[N];

#pragma omp parallel num_threads(N_THREADS)
{
  int thid = omp_get_thread_num();
  int chunk = 2;

  for (int ii = thid * chunk; ii < N; ii+= chunk*N_THREADS) {
    for (int i = ii; i < ii+chunk;  i++) {
           b[i] =  a[i];
    }
  }
}
```

Assume an SMP system with 16 CPUs, each with its own cache memory initially empty. To keep caches coherent the system uses a Snoopy–based write–invalidate MSI coherence protocol. Also assume cache lines of 128 bytes, that `int` size is 4 bytes, first element of vectors `a` and `b` are placed at the first position of memory line and rest of variables are stored in registers. Finally, we know that thread i runs on CPU i.

Although the code above is correct, its execution generates a lot of bus coherence transactions due to false sharing and bad exploitation of spatial locality, and as a consequence, high execution overheads. **We ask you:**

1. Indicate which threads are accessing to the first 6 elements of vector `a` and `b`.

   Solution:

   For both vectors `a` and `b`, elements 0 and 1 are accessed by thread 0, elements 2 and 3 are accessed by thread 1, elements 4 and 5 are accessed by thread 3, etc.

2. Assuming all processor caches are empty at the beginning of the code. Which types of coherence commands are generated by the snoopy controllers when the code is executed in parallel? Just indicate the name of the coherence transactions and which accesses to variables provoke them. Would it be possible to count the number of coherence transactions of each type? Reason your answer.

   Solution:

   Note that first element of vectors `a` and `b` are placed at the first position of memory line and rest of variables are stored in registers. This means that first element of each vector is aligned to memory line (cache line). However, vector a and b are placed in different memory addresses and then, access to one vector doesn't provoke coherence commands affecting the other vector.

Therefore, accesses to vector b provoke BusRdX and Flush transactions[1]. Accesses to vector a only provoke BusRd transactions. For BusRdX and Flush transactions we cannot exactly count the number of bus transactions that will occur since we don't exactly know the order that each thread will access to the elements of vector b. For BusRd transactions we can say that there will be one per chunk of elements accessed ($N/2$ in total).

Details not needed in your answer to the question: In the case of the BusRdx and Flush transactions we can analyze it a little bit more:

- *For BusRdx:*
  - *the minimum number of transactions is one per chunk of elements accessed ($N/2$ in total).*
  - *the maximum number of transactions is the total number of accesses to vector b (N).*
- *For Flush: The number of Flush transactions will be smaller than the number of BusRdX transactions. Note that the first thread writing for the first time to a memory line does not provoke Flush transaction because there is not a dirty copy of this memory line in the rest of the caches. Later, when there exists a previous dirty copy of the memory line, any write to this memory line that provokes a new BusRdX will also provoke a Flush transaction.*

3. Briefly describe the reason why the execution of the code above leads to a a false sharing situation and a lack of spatial locality exploitation. Propose a modification in the code that avoids both situations at once.

   Solution:

   Cache line size is 128 bytes. Each integer is 4 bytes long. Therefore, every 32 consecutive elements of vector b can fit in one cache line (similarly for vector a). In the case of the code, 16 threads, running in different CPUs, access to consecutive elements (in this case each thread access 2 of the 32 elements that fit in a cache line) of vector b (similarly for vector a).

   - In the case of vector b this may provoke a false sharing situation if two or more threads alternatively write to the same cache line.
   - Lack of locality happens because each thread has a cache miss when accessing the first element of each `chunk`, and only can exploit spatial locality for the second element of the chunk.

   To solve both problems, we only need to set the value of `chunk` to 32. This provokes that threads access to different cache lines and fully exploit spatial locality.

[1]BusUpgr transactions are not possible because we are only writing to vector b

Student name: ....

Answer sheet for **Question 3.3**.

Memory access	Affected line	Hit/Miss	CPU command	Bus transaction(s)	Cache line state				NUMA commands	Directory entry	
					0	1	2	3		State	Presence bits
Processor 1 .... x[4]	..	Miss	......	....	I	...	I	I	No	M	....
Processor 2 .... x[16]	..	....	......	....	...	...	S	...	..	..	....
Processor 3 .... x[32]	..	Hit	..... .	$BusUpgr_3$	I	I	I	...	Yes	M	....
Processor 0 .... x[32]	..	....	......	....	M	...	...	...	..	..	....
Processor 2 writes x[20]	..	....	......	....	...	...	...	...	..	..	....