PAR – Final Exam – Course 2021/22-Q1

January 17th, 2022

Problem 1 (2.5 points)

The following code computes matrix `u[N][N]` by blocks of $BS \times N$ elements, with `N` very large:

```
double u[N][N];

// Compute elements in a block of BS x N elements
void compute_row_block(int ii) {
    for (int i=max(1, ii); i<min(ii+BS, N-1); i++)
        for (int j=1; j<N-1; j++) {
            double tmp = u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1] - 4*u[i][j];
            u[i][j] = tmp/4;
        }
}

void main() {
    int NB = 2*P;          // Total number of row blocks
    int BS = N/NB;         // Number of rows per block

    // EVEN loop: traversing all EVEN blocks
    for (int ii=0; ii<NB; ii+=2)
        compute_row_block(ii*BS);

    // ODD loop: traversing all ODD blocks
    for (int ii=1; ii<NB; ii+=2)
        compute_row_block(ii*BS);
}
```

In this code the computation is divided in two parts: the so called *EVEN* loop computing half of the blocks first (blocks 0, 2, ...), and the so called *ODD* loop computing the other half of the blocks later (blocks 1, 3, ...). Before answering the first question below, think about the parallel execution opportunities in this code when defining each iteration of the *EVEN* and *ODD* loops as a task. Could all the tasks in the *EVEN* loop be executed in parallel? Could all the tasks in the *ODD* loop be executed in parallel? And could the execution of tasks in the *ODD* loop be performed in parallel with the execution of tasks in the *EVEN* loop?. Having all that in mind, **we first ask you to:**

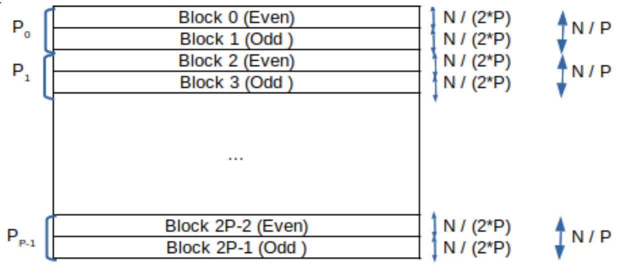
1. Write the expression for the contribution to the total parallel time T_P coming from the parallel computation time T_P^{comp} on an ideal machine with P processors, assuming that: 1) $NB = 2 * P$ perfectly divides the problem size `N`; 2) the execution time for a single iteration of the innermost loop body in routine `compute_row_block` takes t_c time units; and 3) no parallelisation overheads should be considered at this point.

   Solution:

   Since N is very large, we can consider $N - 2 \approx N$. Then,

   $T_P^{comp} = 2 \times BS \times N \times t_c = N^2/P \times t_c$

   Explanation:



All the tasks in the *EVEN* loop can be executed in parallel. All the tasks in the ODD loop can also be executed in parallel. However, tasks in the *ODD* loop can only be executed once the tasks in the EVEN loop computing the surrounding blocks have been completed. Each of the two loops iterate P times. Therefore, with P processors each processor will execute only 1 iteration of each loop. Since N is large, we can consider $N-2 \approx N$. Thus, each processor computes N/P rows of size N, for a total of N^2/P executions of the innermost loop of routine `compute_row_block`. Note that half of them correspond to its *EVEN* block and the other half to its *ODD* block.

Next we consider that the ideal machine has the memory distributed among the P processors, In that machine, matrix `u` is divided row-wise in $NB = 2 * P$ blocks, where each block contains $BS = N/(2 * P)$ consecutive rows. Pairs of consecutive blocks are assigned to the same processor, so for example blocks 0 and 1 are owned by processor 0, blocks 2 and 3 are owned by processor 1; and so on and so forth. Each processor is in charge of computing all the rows in the blocks that are assigned to it, and therefore it will have to execute one iteration of the *EVEN* loop and one iteration of the *ODD* loop. Before answering the second question below, and having in mind the assignment of blocks and iterations, think about the need for processors to perform any remote access before starting the execution of tasks in the *EVEN* loop, or before starting the execution of tasks in the *ODD* loop; and, if affirmative, how many elements need to be transferred in each case and which processors to do them. Having all that in mind, next **we ask you to:**

2. Write the expression for the contribution to T_P coming from the data sharing overheads $T_P^{data\_sharing}$, if any, as part of the overall $T_P = T_P^{comp} + T_P^{data\_sharing}$. For that you should consider the data sharing model explained in class. Accesses to local memory are performed with zero overhead; accesses to remote memory take $t_{comm} = t_s + m \times t_w$, being t_s and t_w the start–up time for the remote access and transfer time of one element, respectively. At a given moment, a processor can only perform one remote memory access to another processor, and can only serve a remote memory access from another.

   Solution:

   $T_P^{data\_sharing} = 2 \times (t_s + N \times t_w)$

   Explanation:

   In general, before an EVEN block can be computed a processor needs to perform a remote access to the N elements in the last row of the block above, which is owned by the previous processor (except for processor 0). No remote access is required to access the first row of the next block, because it is an ODD block owned by the same processor. All the processors can perform such remote accesses in parallel.

   An ODD block can only be computed once its surrounding EVEN blocks are computed. Before an ODD block can be computed a processor needs to perform a remote access to the N elements in the first row of the block below, which is owned by the next processor. No remote access is required to access the last row of the previous block, because it is an EVEN block owned by the same processor. All the processors can perform such remote accesses in parallel.

Problem 2 (2.5 points)

Given the following code fragment:

```
#define CACHE_LINE_SIZE  128
#define DBSize           (32*1024*1024)
#define NTHREADS         3

int count[NTHREADS];
int DB[DBSize];
int key;

// Initialization loop
for (int i = 0; i < NTHREADS; i++)
    count[i] = 0;
```

```
#pragma omp parallel num_threads(NTHREADS)
{
    int my_id = omp_get_thread_num();
    for (int i = my_id; i < DBSize; i += NTHREADS) {
        // read access to DB[i]
        if (DB[i]==key)
            // read access to count[my_id] followed by a write access
            count[my_id] = count[my_id] + 1;
    }
}
```

Assume a shared–memory UMA parallel system with 3 processors, each one with its own (private) cache memory. Data coherence in the system is maintained using Write Invalidate `MSI` protocol, with a Snoopy attached to each cache memory. Also assume 1) empty caches at the beginning of the program; 2) `count[0]` and `DB[0]` are aligned to the beginning of different memory lines; 3) the rest of variables are stored in registers; and 4) the size for a variable of type `int` is 4 bytes and cache line size is 128 bytes. **We ask you:**

1. Assume thread 0, running on processor 0, starts executing the sequential part of the program until the parallel region starts (i.e. it executes the initialization loop). How many cache lines are used to store the full `count` vector after the execution of the initialization loop? What is the cache coherence state for each cache line storing elements of `count` and in which private cache it is located?

   Solution:

   We only require one cache line to store the three elements of the vector `count`. The coherence state of the cache line will be M (Modified) and the memory cache that will store the vector is MC_0.

2. Assume the cache states in previous question (after the execution of the initialization loop) and that each thread i runs on processor P_i of the UMA system within the parallel region. Complete the table in the provided answer sheet which represents the first sequence of actions (from top to bottom of the table) executed by the three threads in the parallel region for the first iterations of the loop. State MC_i in each row has to be filled with the state of the cache line after the memory access in the action. For the rest of columns, you should specify the processor event ($PrRd_i$, $PrWr_i$), if there is cache hit or miss, the bus command ($BusRd_i$, $BusRdX_i$, $BusUpgr_i$), and the flush transaction ($Flush_i$), where i is the number of the processor where it is generated. Note: Observe that if you want to leave a cell empty, you can write "-".

   Solution:

Parallel Region Execution							
Action	CPU event	Miss/Hit	Bus command	Flush?	State MC_0	State MC_1	State MC_2
P_0 read DB[0]	$PrRd_0$	miss	$BusRd_0$	-	S	-	-
P_0 read count[0]	$PrRd_0$	hit	-	-	M	-	-
P_1 read DB[1]	$PrRd_1$	miss	$BusRd_1$	-	S	S	-
P_1 read count[1]	$PrRd_1$	miss	$BusRd_1$	$Flush_0$	M->S	S	-
P_0 write count[0]	$PrWr_0$	hit	$BusUpgr_0$	-	S->M	S->I	-
P_1 write count[1]	$PrWr_1$	miss	$BusRdX_1$	$Flush_0$	M->I	I->M	-
P_2 read DB[2]	$PrRd_2$	miss	$BusRd_2$	-	S	S	S
P_2 read count[2]	$PrRd_2$	miss	$BusRd_2$	$Flush_1$	I	M->S	S
P_2 write count[2]	$PrWr_2$	hit	$BusUpgr_2$	-	I	S->I	S->M
P_0 read DB[3]	$PrRd_0$	hit	-	-	S	S	S
P_1 read DB[4]	$PrRd_1$	hit	-	-	S	S	S

3. Assuming the amount of data we are accessing in the program, a system with a main memory of 32 GBytes and 32 Kbytes of private cache memory per processor, what is the total number of bits that an UMA system would use to keep the cache coherence per processor and the overall system? Would this number of bits change if we change the protocol from `MSI` to `MESI`?

   Solution:

   Per processor/cache:
   $32kbytes \times \frac{1line}{128bytes} \times \frac{2bits}{1line} \rightarrow \frac{2^{15}\times 2}{2^7} bits \rightarrow 2^9 bits \rightarrow 512 bits$

   Overall:
   3×512 bits devoted to coherence

   The number of bits would not change because we can still use 2 bits to represent 4 states (M, E, S, I) as when we have 3 states (M, S, I).

Problem 3 (2.5 points)

Given the following sequential program that computes the number of times the value stored in variable `element` appears in a vector of lists `data`:

```
#define NUM_ELEMS 10000
typedef struct list {
    int elem;
    struct list * next;
} list; // the basic component of a list

list * data[NUM_ELEMS]; // vector of lists, each list with varying number of elements
int element, count = 0; // value to search within data and number of times it appears

// function that returns the number of times element appears in theList
int list_search(list * theList, int element);

void main() {
    ...
    for (int entry = 0; entry < NUM_ELEMS; entry++) {
         int tmp = list_search(data[entry], element);
         count += tmp;
    }
    ...
}
```

Each of the lists may have a different number of elements, so when parallelising the program one should take care of load balancing. A parallel version for the sequential program above is also available, in which the original `for` loop has been substituted by a parallel region that assigns individual iterations to explicit tasks in such a way that tasks are generated under certain circumstances. One new shared variable `active_tasks` and two functions to operate it have also been added:

```
...
int active_tasks = 0;
// Functions to atomically add or subtract 1 to memory location pointed by address.
// The operation is saturated to the max or min value (i.e. the result can not be
// greater than max and smaller than min, respectively). They return value in memory
// location before operation
int atomic_inc(int *address, int max);
int atomic_dec(int *address, int min);

void main() {
    #pragma omp parallel
    #pragma omp single
    {
        int workers = omp_get_num_threads() - 1; // one thread focuses on
                                                 // task creation, the rest execute tasks
        for (int entry = 0; entry < NUM_ELEMS; entry++) {
            while (atomic_inc(&active_tasks, workers) == workers);
            #pragma omp task depend(inout: count)
                {
                int tmp = list_search(data[entry], element);
                count += tmp;
                atomic_dec(&active_tasks, 0);
                }
        }
    }
}
```

After compiling the parallel program and executing it with P processors (with $P > 1$) we detect that the program **is not achieving any speed–up**, although it produces a correct result.

1. Assuming the functionality for functions `atomic_inc` and `atomic_dec` explained in the code itself, what are these two functions used for in the program? Which is the number of implicit and explicit tasks that are generated during the execution of the program?

   Solution:

   The program generates P implicit tasks, one for each thread in the `parallel` construct. Only one of them (the one entering in `single`) is in charge of generating the explicit tasks: one explicit task is generated for each iteration of the for loop, so in total `NUM_ELEMS` explicit tasks. Functions `int atomic_inc(int *address, int max)` and `int atomic_dec(int *address, int min)` are used to control the number of tasks generated (kind of cut-off mechanism), in such a way that no more than $P - 1$ explicit tasks are simultaneously pending to execute or executing.

2. In the parallel version provided above, how many of these explicit tasks can be simultaneously executing? Rewrite the program above making the minimum appropriate changes in order to increase this number and, as a consequence, achieve a much better speed–up. Make sure the program generates the correct result. After these minimal changes, which can be the maximum number of explicit tasks that could be simultaneously executing?

   Solution:

   The `depend(inout:count)` clause used in `task` forces explicit tasks to execute sequentially, in the order they are created. This is not the most appropriate way to guarantee the race condition in this program when updating variable `count`; instead the use of `atomic` would enable the parallelism in the execution of multiple invocations to `list_search` while guaranteeing the correct update of variable `count`. With this change, a maximum of $P - 1$ tasks could be executing simultaneously.

   ```
   #pragma omp task // shared(count) and firstprivate(entry) by default
       {
       int tmp = list_search(data[entry], element);
       #pragma omp atomic
       count += tmp;
       atomic_dec(&active_tasks, 0);
       }
   ```

   Alternatively, a solution based on task reductions would also be valid:

   ```
   #pragma omp taskgroup task_reduction(+: count)
   for (int entry = 0; entry < NUM_ELEMS; entry++) {
       while (atomic_inc(&active_tasks, workers) == workers);
   ```

```
            #pragma omp task in_reduction(+: count)
                {
                int tmp = list_search(data[entry], element);
                count += tmp;
                atomic_dec(&active_tasks, 0);
                }
        }
```

3. Do an implementation for function `atomic_inc` making use of load–linked (`ll`) and store–conditional (`sc`) operations, defined as follows:

```
int ll(int *address); // returns the value stored in address
int sc(int *address, int value); // stores value in address if atomicity with ll
                                 // has been accomplished, returning true (1);
                                 // retuns false (0) otherwise
```

Solution:

A possible solution could be:

```
int atomic_inc(int *address, int max) {
    int tmp = ll(address);
    while ((tmp < max) && !sc(address, tmp+1)) tmp = ll(address);
    return(tmp);
}
```

In this solution, if the value read from memory `address` is equal to `max`, then the while loop finishes, returning the value just read. If the value is smaller than `max`, then a `sc` of the incremented value on the same memory `address` is attempted; if it succeeds the while loop is finished, again returnin the original value read from memory. If `sc` fails, then the new value from memory `address` is read again.

An equivalent code written in a different (more explicit) way would be:

```
int atomic_inc(int *address, int max) {
    int retsc = 0;
    do {
        int tmp = ll(address);
        if (tmp == max) return(tmp);
        retsc = sc(address, tmp+1);
    } while (retsc == 0);
    return(tmp);
}
```

Another version that always writes to memory `address` would be:

```
int atomic_inc(int *address, int max) {
    int tmp, newvalue;
    do {
        tmp = newvalue = ll(address);
        if (tmp < max) newvalue++;
    } while (!sc(address, newvalue));
    return(tmp);
}
```

Observe that in this case if the value in memory `address` is already `max`, the same value `max` is unnecessarily written again to memory `address`.

4. Finally we found a recursive version alternative to the original sequential code:

```
...
void rec_list_search(list ** data, int size, int element) {
    int tmp = list_search(data[0], element);
    count += tmp;
    if (size > 1)
        rec_list_search(data+1, size-1, element);
}

void main() {
    rec_list_search(&data[0], NUM_ELEMS, element);
}
```

   Write a parallel version for it that implements a *recursive* **leaf** *task decomposition.* This new version should not implement any cut-off mechanism to restrict the number of tasks that are generated.

   Solution:

```
void rec_list_search(list ** data, int size, int element) {
    #pragma omp task shared(count)
    {
        int tmp = list_search(data[0], element);
        #pragma omp atomic
        count += tmp;
    }
    if (size > 1)
        rec_list_search(data+1, size-1, element);
}

void main() {
    ...
    #pragma omp parallel
    #pragma omp single
    rec_list_search(&data[0], NUM_ELEMS, element);
    ...
}
```

Problem 4 (2.5 points)

Given the following code fragment computing matrix `m[N][N]`, with `N` much larger than the number of processors P to be used in the parallel execution, and with `N` not necessarily a multiple of P:

```
telem m[N][N];

for (int i=1; i<N-1; i++)
   for (int j=0; j<N; j++)
       m[i][j] = compute (m[i][j], m[i-1][j], m[i+1][j]);
```

We ask you to:

1. Decide the most appropriate *geometric data decomposition strategy* for matrix `m` and write a parallel version of the code above using OpenMP that corresponds to it. Your solution should a) minimize the synchronization overhead among implicit tasks 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).

   Solution:

   There are dependencies between iterations of the `for-i` loop: a RAW dependency given by $m[i][j]$ to $m[i-1][j]$ and a WAR dependency given by $m[i][j]$ to $m[i][j+1]$ for i in $[2..N-2]$. There are no dependencies between iterations of the `for-j` loop, so we can fully parallelize it, with no need for synchronization.

Consequently, it's advisable to choose a *Geometric Block Data decomposition* by columns. We must take care of adjusting the block size (number of columns of N elements each) so that there is at most 1 column of difference in size between the different blocks. In order to apply the owner compute rule, the distribution of the matrix `m` is by columns with the resulting block size. A *Geometric Cyclic Data decomposition* by columns would also be correct, as the amount of work would be balanced automatically, without any extra calculation.

```
telem m[N][N];
...
#pragma omp parallel num_threads(P)
{
  int myid = omp_get_thread_num();
  int BS = N / P;
  int start = myid * BS;
  int end = start + BS;
  int mod = N % P;
  if (mod > 0) {
     if (myid < mod)
        start += myid;
        end = start + BS + 1;
     }
     else {
         start += mod;
         end += start + BS;
     }
  }
  for (int i=1; i<N-1; i++) {
    for (int j=start; j<end; j++) {
        m[i][j] = compute (m[i][j], m[i-1][j], m[i+1][j]);
    }
  }
}
```

2. Now consider that the program is going to be executed on a parallel machine in which memory lines are 128 bytes long. The allocation in memory for matrix `m` is aligned to the start of a memory line and `sizeof(telem)` is 8 bytes (`N` is not necessarily multiple of `sizeof(telem)`). Decide the most appropriate *geometric data decomposition strategy* in this case and re-write the previous OpenMP parallel code and, if necessary, the definition of matrix `m`. Your solution should a) maximize parallelism among implicit tasks; and b) maximize data locality and reduce coherence traffic.

   Solution:

   In order to preserve data locality we must take care of the memory line size when accessing elements of the matrix. In this sense, given that `sizeof(telem) = 8 bytes`, a memory line of 128 bytes can hold up to 16 elements. For this reason we will consider only block sizes with values multiple of 16. Consequently we apply a *Geometric Block-cyclic data decomposition* by columns, with block size = 16. Given that N » P we can assume a load unbalance of one block (N x 16 elements). In addition, in case N is not multiple of `sizeof(telem)`, we must add padding at the end of the row to avoid generating false sharing with the beginning of the next row. In order to apply the owner compute rule, the distribution of the matrix `m` is by columns with the resulting block size.

```
#define MEMORYLINESIZE 128
#define BS MEMORYLINESIZE/sizeof(telem)
#define X (N%BS==0? 0: (BS - (N % BS)))
```

```
telem m[N][N+X];

int BS = MEMORYLINESIZE / sizeof(telem);
...
#pragma omp parallel num_threads(P)
{
  int myid = omp_get_thread_num();
  int start = myid * BS;
  int end = N;

  for (int i=1; i<N-1; i++) {
    for (int jj=start; jj<end; jj+=BS*P)
        for (int j=jj; j<j+BS; j++)
            m[i][j] = compute (m[i][j], m[i-1][j], m[i+1][j]);
}
```

Student name: ....

Answer sheet for **Question 2.2**.

Parallel Region Execution							
Action	CPU event	Cache Miss/Hit	Bus command	Flush?	State MC_0	State MC_1	State MC_2
P_0 reads DB[0]							
P_0 reads count[0]							
P_1 reads DB[1]							
P_1 reads count[1]							
P_0 writes count[0]							
P_1 writes count[1]							
P_2 reads DB[2]							
P_2 reads count[2]							
P_2 writes count[2]							
P_0 reads DB[3]							
P_1 reads DB[4]							