PAR – Final Exam – Course 2020/21-Q2
June 16th, 2021

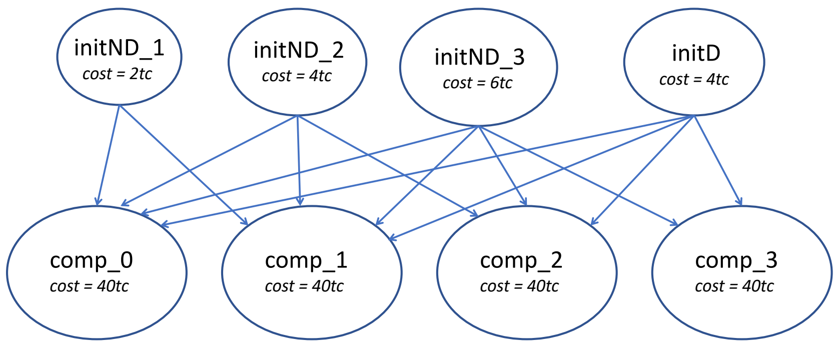
Problem 1 (2 points) Given the following nested loops in a C program instrumented with *Tareador*:

```
 #define N 4
 int A[N][N], B[N][N];
 ...
 // initialization of non-diagonal elements
 for (i=1; i<N; i++) {
     sprintf(stringMessage, "initND_%d", i);
     tareador_start_task (stringMessage);
     for (k=0; k<i; k++) {
         A[i][k] = init(i,k); // inner loop body cost = 2*tc
         A[k][i] = A[i][k];
     }
     tareador_end_task (stringMessage);
}

// initialization of diagonal elements
tareador_start_task ("initD");
for (i=0; i<N; i++) A[i][i] = init (i,i); // inner loop body cost = 1*tc
tareador_end_task ("initD");

// computation phase
for (i=0; i<N; i++) {
    sprintf(stringMessage, "comp_%d", i);
    tareador_start_task (stringMessage);
    for (k=0; k<N; k++) B[i][k] = foo (A[i][k]);  // inner loop body cost = 10*tc
    tareador_end_task (stringMessage);
}
```

We ask you to answer the following questions:

1. Draw the Task Dependence Graph (TDG) assuming the given value for constant N and the *Tareador* task definitions in the program. In the TDG, annotate each node with the name of the corresponding tasks (initND_i, initD, comp_i) and its cost.

   Solution:



2. Compute the T_1, T_∞ and P_{min} metrics associated to the TDG obtained in the previous question.

   Solution:

   The sum up of the cost of all the tasks determines $T_1 = 176 \times tc$. The critical path is composed of nodes: initND_3 and comp_X, where X is any number between 0 and 3. Its cost is $T_\infty = 46 \times tc$. The minimum number of processors to achieve T_∞ execution time is $P_{min} = 4$.

3. Determine the assignment of tasks to processors that would yield the best *speed-up* on 4 processors. Calculate T_4 and S_4.

   Solution:

   Since the number of requested preocessors is $P = P_{min} = 4$, then $T_4 = T_\infty = 46 \times tc$, and therefore $S_4 = 176 \ / \ 46 = 3.8$ coincides with the *Parallelism* metric. The assignment of tasks to 4 processors is straightforward:

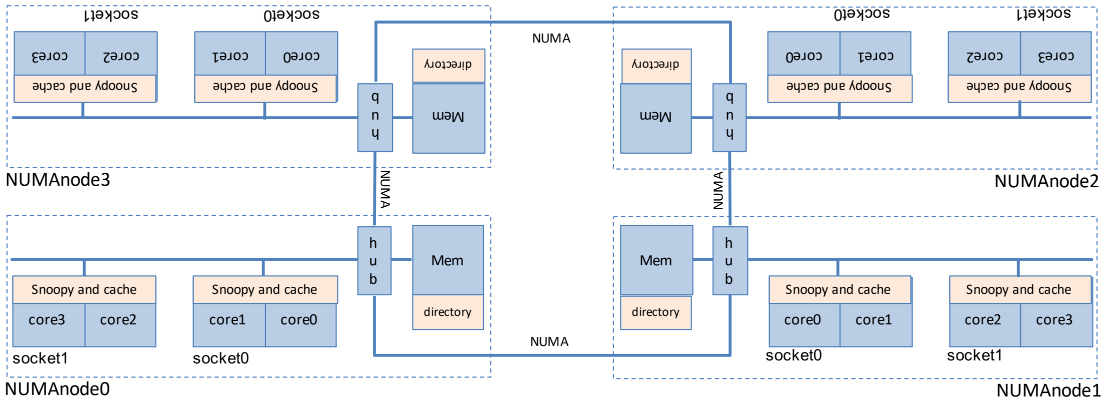
   P_0 = `initND_1, comp_0`

   P_1 = `initND_2, comp_1`

   P_2 = `initND_3, comp_2`

   P_3 = `initD, comp_3`

Problem 2 (1 point) Given the following NUMA system with 4 NUMA nodes, each NUMA node with 2 sockets sharing the access to node memory, and each socket with two cores sharing the access to a per-socket cache. Coherence inside NUMA nodes is maintained with a snoopy–based mechanism implementing the simplest write–invalidate MSI explained in class. Coherence among NUMA nodes is maintained with a directory–based mechanism implementing the simplest write–invalidate MSU explained in class.



Assume that the home node for the line containing variable `var` is NUMAnode0, and at a given time there exist 3 clean copies of that line in cache memories: in socket0 in NUMAnode0, in socket0 in NUMAnode1 and in socket0 in NUMAnode2. Considering that the following memory accesses are done one after the other: 1) core2 in NUMAnode0 reads `var`; 2) core0 in NUMAnode3 reads `var`; and 3) core0 in NUMAnode3 writes `var`. **We ask you to select ONLY the eight sentences that you consider correct from the list below** (labeled from a) to o)). Each correct selection adds 0.125 points; each wrong selection subtracts 0.0625 points; if you select more than 8, only the first 8 will be considered; the grade for this problem is always in the range 0–1.

1. When core2 in NUMAnode0 reads variable `var`, which of the following sentences are correct?

   (a) Core2 issues PrRd.

   (b) The snoopy in socket1 issues BusRd on its local bus.

   (c) The snoopy in socket0 observes the BusRd command and places the line on the bus (Flush).

   (d) The hub associated to NUMAnode0 updates the directory for the line containing `var` to indicate that a new copy of the line exists inside NUMAnode0.

   (e) No coherence requests are sent to the rest of the NUMA nodes in the system.

   Solution: True, True, False, False, True

2. Then, when core0 in NUMAnode3 reads variable `var`, which of the following sentences are correct?

    (f) The snoopy in socket0 issues BusRd on its local bus.

    (g) The hub associated to NUMAnode3 finds the closest NUMA node that has a copy of the line and sends a RdReq to that NUMA node.

    (h) The hub of the NUMA node receiving the RdReq reads the line from the cache memory that is storing it.

    (i) NUMAnode3 receives a Dreply command with the line containing variable `var` and stores a copy in its main memory.

    (j) At the end the directory in the home NUMA node is updated so that all bits in the sharers list are set to 1 and the state is kept as S

    Solution: True, False, False, False, True

3. Finally, when core0 in NUMAnode3 writes variable `var`, which of the following sentences are correct?

    (k) The snoopy in socket0 of NUMAnode3 issues an Invalidate command on its local bus.

    (l) The hub in NUMAnode3 issues an Invalidate command, going to the home NUMA node.

    (m) The home NUMA node checks if there are copies of the line in other NUMA nodes, sending to each one of them an Invalidate command.

    (n) The state for the line in the caches of the remote nodes receiving the Invalidate command (as well as in the home node) changes from S to I to indicate that the line is nor valid anymore.

    (o) At the end the directory in the home NUMA node only has bit 3 in the sharers list set to 1 and the state set to M.

    Solution: False, False, True, True, True

Problem 3 (3 points) Assume the following sequential code and $N >= 2$ and power of two:

```
#define N (1<<29) // A power of 2 value
typedef struct {
  float max; float min;
} min_max_t;

min_max_t find_min_max_it(float *v, int n) {
     min_max_t min_max;
     float max_duration = v[1]-v[0];
     float min_duration = v[1]-v[0];

     for (int i=2; i<n; i++) {
        float d = v[i] - v[i-1];
        if (d > max_duration) max_duration = d;
        if (d < min_duration) min_duration = d;
     }

     min_max.max = max_duration;
     min_max.min = min_duration;
     return min_max;
}

min_max_t find_min_max_rec(float *v, int n) {
    min_max_t min_max, min_max1, min_max2;

    if (n==2) {
        min_max.max = v[1]-v[0]; min_max.min = v[1]-v[0];
    } else {
        int n2 = n/2; // n is power of 2
        min_max1 = find_min_max_rec(v, n2);
        min_max2 = find_min_max_rec(v+n2, n2);

        if (min_max1.min < min_max2.min) min_max.min = min_max1.min;
        else min_max.min = min_max2.min;
        if (min_max1.max > min_max2.max) min_max.max = min_max1.max;
        else min_max.max = min_max2.max;
    }
    return min_max;
}

int main() {
    min_max_t min_max_V1, min_max_V2;
    int v1[N], v2[N];
    min_max_V1 = find_min_max_it(v1, N);
    min_max_V2 = find_min_max_rec(v2, N);
}
```

We ask you to answer the following independent questions:

1. Implement an OpenMP parallel version of function `find_min_max_it` and modify the main program as you consider to create an efficient iterative linear task decomposition version of the code. This implementation should avoid synchronizations within the loop and exploit the parallelism with a grainsize bigger than one iteration per task. You are ONLY allowed to use explicit tasks.

   Solution:

   The key points are :

   - Usage of taskloop construct with a granularity bigger than 1, for instance, 2 (`grainsize(2)`) or evenly distributing the iterations among threads (`num_tasks(omp_get_num_threads())`).
   - Usage of reduction clause to avoid synchronizations to update the max and min durations.
   - Add constructs parallel and single in the main program.

```
typedef struct {
  float max; float min;
} min_max_t;

min_max_t find_min_max_it(float *v, int n) {
     min_max_t min_max;
     float max_duration = v[1]-v[0];
     float min_duration = v[1]-v[0];

     /* Assuming the number of threads is smaller than n */
     #pragma omp taskloop num_tasks(omp_get_num_threads()) \
                          reduction(max:max_duration)      \
                          reduction(min:min_duration)
     for (int i=2; i<n; i++) {
        float d = v[i] - v[i-1];
        if (d > max_duration) max_duration = d;
        if (d < min_duration) min_duration = d;
     }

     min_max.max = max_duration;
     min_max.min = min_duration;
     return min_max;
}
```

```
int main() {
    min_max_t min_max_V1, min_max_V2;
    float v1[N], v2[N];

    #pragma omp parallel
    #pragma omp single
    min_max_V1 = find_min_max_it(v1, N);

    min_max_V2 = find_min_max_rec(v2, N);
    }
}
```

2. Implement an OpenMP parallel version of function `find_min_max_rec` and modify the main program as you consider to create an efficient recursive task decomposition version of the code. This implementation should reduce the parallelization overheads due to the generation of tasks controlling it by the depth of the recursivity tree (MAX_DEPTH).

   Solution:

   The key points are :

   - Implement a recursive tree task decomposition
   - Add a new parameter to count the depth level
   - Usage of final and mergeable clauses to implement the cut-off based on the recursivity tree level. Note that it can be implemented using if statements controlling if the maximum depth is reached or not.
   - Force `min_max1` and `min_max2` to be shared, and a taskwait to wait for the two created tasks to be finished.
   - Add constructs parallel and single in the main program.

```
typedef struct {
  float max; float min;
} min_max_t;

min_max_t find_min_max_rec(float *v, int n, int depth) {
    min_max_t min_max, min_max1, min_max2;

    if (n==2) {
        min_max.max = v[1]-v[0]; min_max.min = v[1]-v[0];
    } else {
        int n2 = n/2; // n is power of 2

        #pragma omp task shared(min_max1) final(depth>=MAX_DEPTH) mergeable
        min_max1 = find_min_max_rec(v, n2, depth+1);

        #pragma omp task shared(min_max2) final(depth>=MAX_DEPTH) mergeable
        min_max2 = find_min_max_rec(v+n2, n2, depth+1);

        #pragma omp taskwait

        if (min_max1.min < min_max2.min) min_max.min = min_max1.min;
        else min_max.min = min_max2.min;
        if (min_max1.max > min_max2.max) min_max.max = min_max1.max;
        else min_max.max = min_max2.max;
    }
    return min_max;
}
```

```
    int main() {
        min_max_t min_max_V1, min_max_V2;
        int v1[N], v2[N];
        min_max_V1 = find_min_max_it(v1, N);

        #pragma omp parallel
        #pragma omp single
        min_max_V2 = find_min_max_rec(v2, N, 0);
    }
```

Problem 4 (4 points) Assume the following sequential code fragment implementing a certain computation with matrix `out_matrix` and vector `in_vector`:

```
#define N ... // number of elements in the input vector
#define M ... // number of rows and columns in the output matrix
double in_vector[N];
double out_matrix[M][M];
...
int i, row;
...
for (i = 0; i < N; i++) {
    row = random(M); // random returns a random number between 0 and M-1
    update_row(row, in_vector[i]);
}
```

The following code implements a parallel version for the above loop that uses the so called *"master–worker" paradigm*. In the *"master–worker" paradigm* the "master" thread (only one, thread P in the code below) is the only responsible for assigning work to the "worker" threads (P threads, numbered from 0 to P-1 in the code below, assuming a parallel region executed with P+1 processors). Communication between the "master" thread and a "worker" thread `k` is done through one element of vector `port`, in particular `port[k]`. Through this port `port[k]` the master sends to worker `k` the rows that it has to compute, one after the other, following a specific **output data decomposition** strategy:

```
#define N ... // number of elements in the input vector
#define M ... // number of rows and columns in the output matrix
#define P ... // number of worker threads
double in_vector[N];
double out_matrix[M][M];

typedef struct {
    int row;
    double value;
} Port;
Port port[P];

int i, row, destination;
...
#pragma omp parallel num_threads(P+1)
if (omp_get_thread_num() == P) {
    for (i = 0; i < N; i++) {
        row = random(M); // random returns a random number between 0 and M-1
        destination = thread_to_be_assigned(row, M, P);     // Question 4.1
        port[destination].row= row;
        port[destination].value= in_vector[i];
    }
} else {
    myid = omp_get_thread_num();
    for ( ; ; ) {
        update_row(port[myid].row, port[myid].value);
    }
}
```

The previous code is not complete since the master and worker threads need some sort of synchronization to ensure the proper assignment of work from master to worker threads. However, you SHOULD NOT WORRY about this issue by now and will address it later.

We ask you to:

1. Implement 3 versions of function `int thread_to_be_assigned(int row, int num_rows, int num_procs)` to implement a:
   (a) *BLOCK* data decompositon, assuming that M is a multiple of P;
   Solution:
   ```
   int thread_to_be_assigned(int row, int num_rows, int num_procs) {
       int num_elems = num_rows / num_procs;
       return (row / num_elems);
   }
   ```
   (b) *CYCLIC* data decomposition;
   Solution:
   ```
   int thread_to_be_assigned(int row, int num_rows, int num_procs) {
       return (row % num_procs);
   }
   ```
   (c) *BLOCK-CYCLIC* data decompositon, with block size `BS`;
   Solution:
   ```
   #define BS ...
   int thread_to_be_assigned(int row, int num_rows, int num_procs) {
       return ((row / BS) % num_elems);
   }
   ```

To address the synchronization issue between master and worker threads mentioned before the programmer is proposing to add a new field `ready` to the definition of `Port`, initially set to 0, and two new functions `wait4worker` and `wait4master`, as follows:

```
typedef struct {
    int ready;
    int row;
    double value;
} Port;

Port port[P];

void wait4worker (int num) {
    while (port[num].ready == 1);
    port[num].ready = 1;
}

void wait4master (int num) {
    while (port[num].ready == 0);
    port[num].ready = 0;
}
```

2. Modify the implementation of function `wait4worker` so that its execution is performed atomically (i.e. the read/write of `port[num].ready` is performed atomically), making use of the following atomic primitive:

   - `int test_and_set(int *addr)`: returns the value stored at the memory address pointed by `addr` and sets it to 1;

   Solution:

```
void wait4worker (int num) {
    while (test_and_set(&port[num].ready) == 1);
}
```

   Of course, solutions implementing a *test-test&set* solution have also been considered valid.

3. Similarly, modify the implementation of function `wait4master` so that its execution is performed atomically (i.e. ensuring atomicity in the read/write of `port[num].ready`), making use of the following atomic primitives:

   - `int load_linked (int *addr)`: returns the value stored at the memory address pointed by `addr`;
   - `int store_conditional (int *addr, int value)`: tries to write `value` into the memory address pointed by `addr`, returning 1 if it succeeds (no intervening store to that address has taken place since the last call to `load_linked` with the same memory address) or 0 if it fails.

   Solution:

```
void wait4master (int num) {
    do {
        while (load_linked(&port[num].ready) == 0);
    } while (store_conditional(&port[num].ready, 0) == 0);
}
```

You can assume that functions `test_and_set`, `load_linked` and `store_conditional` are compatible in terms of atomicity. **You do not have to modify the original parallel code to make it correct using functions `wait4worker` and `wait4master`, you simply need to implement an atomic version of these two functions.**

Finally, the programmer wants to avoid the possibility of having false sharing when accessing vector `port`.

4. Why false sharing may happen when accessing to vector `port`? Redefine the last definition of data structure `Port` to ensure that false sharing will not occur, assuming that `int` and `double` data types occupy 4 and 8 bytes, respectively, and that cache and memory lines are 64 bytes long,

   Solution: False sharing may occur since several consecutive elements of vector `port` can reside in the same cache line and each of them read/written by a different processor. The solution would be to make sure each element occupies a complete cache line. Since the structure `Port` includes 2 integer and 1 double, in total 16 bytes, we can add padding for a total of $64 - 16$ bytes, that is 48 bytes (which are occupied for example by 12 integer elements):

```
typedef struct {
    int ready;          // 4 bytes
    int row;            // 4 bytes
    double value;       // 8 bytes
    int padding[12];    // 48 bytes
} Port;                 // 64 bytes

Port port[P];           // 64 bytes per element
```

Another option would be to convert `port` to a matrix, so that each row occupies a complete cache line. To achieve that, since the structure occupies 16 bytes, we need 4 elements in each row:

```
typedef struct {
    int ready;          // 4 bytes
    int row;            // 4 bytes
    double value;       // 8 bytes
} Port;                 // 16 bytes

Port port[P][64/16];    // 64 bytes per row
```

and then modify all accesses in the code accordingly to only access to column 0, for example: `port[destination][0]` `row`.