PAR – Final Exam – Course 2022/23-Q1

January 18^{th}**, 2023**

Problem 1 (2.5 points)

Given the following code with *tareador* task annotations:

```
#define p ...   // Number of processors
#define NR ...  // Number of rows
#define NC ...  // Number of columns

int M[NR][NC];
int BS = NR/p;  // Assume p divides NR exactly

for (int ii=0; ii<NR; ii+=BS) {                  // Matrix initialization
    tareador_start_task ("init");
    for (int i=ii; i<ii+BS; i++)
       for (int j=0; j<NC; j++)
         M[i][j] = init(i,j);                    // <-- Cost ti
    tareador_end_task ("init");
}

tareador_start_task ("comp1");                   // Begin computations
for (int i=0; i<BS; i++)
    for (int j=0; j<NC; j++)
        M[i][j] = comp1(M[i][j]);                // <-- Cost tb
tareador_end_task ("comp1");

for (int ii=BS; ii<NR; ii+=BS) {                 // Final computations
    tareador_start_task ("comp2");
    for (int i=ii, int i0=0;
         i<ii+BS;
         i++, i0++)
       for (int j=0; j<NC; j++)
         M[i][j] = comp2(M[i0][j], M[i][j]);     // <-- Cost tf
    tareador_end_task ("comp2");
}
```

Let us assume the data sharing model explained in class based on a distributed memory architecture in which we consider that local memory accesses have no cost, but an access to data in different processors introduces a data–sharing overhead: the access time to remote data is determined by $t_{comm} = t_s + m \times t_w$, being t_s and t_w the "start-up" and sending time of an element, respectively, and being m the size of the message. Also, according to the data sharing model: at any time, each processor can simultaneously make one remote access to a different processor and serve one remote access from another processor.

Assume that the number of processors p divides the number of rows `NR` exactly; routines `init`, `comp1` and `comp2` do not modify any memory position; the execution time of one iteration of the body of the most internal loops is t_i, t_b and t_f respectively; tasks are scheduled to processes following the *owner-computes rule* so that a task will be executed by the processor who owns the memory that holds the output of that task; the matrix is already distributed in the memory of each processor when the computation starts; the resulting matrix remains distributed and there is no final communication to a single processor; the strategy used for decomposing matrix `M` follows a *block row distribution*: the matrix `M` is distributed so that each processor has $\frac{NR}{p}$ consecutive rows. **We ask** you to:

1. Draw a Task Dependence Graph (TDG) for the case where $p = 4$;
2. Draw a timeline for the execution with $p = 4$;
3. Provide a general expression that determines the parallel execution time on p processors (T_p): express T_p as a function of p, `NR`, `NC`, t_i, t_b, t_f, t_s and t_w.

Problem 2 (2.5 points)

Assume a NUMA (non-uniform memory architecture) multiprocessor system composed of 4 NUMA-nodes ($node_{0:3}$), each node with 8 GBytes of main memory and 4 cores ($core_{0:3}$). Each core has only one level of cache memory of 2 MBytes (with cache lines of 32Bytes). The system includes all the necessary mechanisms (seen in class) to keep the memory coherence inside a NUMA-node and between NUMA-nodes.

1. (0.5 points) In order to support a write-invalidate MSI cache-coherence protocol within each NUMA-node, how many additional bits should be used in each cache line, and what are their role and possible values? How many bits in total per NUMA-node (only within each NUMA-node) are used for that purpose?

2. (0.5 points) In order to support a directory-based write-invalidate MSU cache-coherence protocol between NUMA-nodes, how many bits should be used in the directory for each line in main memory, and what are their role and possible values? How many bits in total per NUMA-node (directory) are used for that purpose?

3. (1.5 points) Consider the following parallel program executed on only two cores (processors) inside $node_0$ of the previous multiprocessor system:

```
...
double A[M][N];
...
```

Core 0: Update even-numbered rows

```
for ( j = 0 ; j < M ; j += 2 )
   for ( k = 0 ; k < N ; k++ )
        A[j][k] = f(j,k);
```

Core 1: Update odd-numbered rows

```
for ( j = 1 ; j < M ; j += 2 )
   for ( k = 0 ; k < N ; k++ )
        A[j][k] = g(j,k);
```

Assume that matrix `A` is stored in main memory of $node_0$ (without copies of any of its elements in the caches of the NUMAnodes), that each element of matrix `A` is 8 Bytes and that the first element of `A` is aligned on a cache line boundary, $N = 2$ and M is a value multiple of 2.

(a) What would be the maximum number of invalidations that would be sent through the bus expressed as a function of M? What is causing such a memory coherence problem?

(b) Modify the declaration of matrix A so that you do not have to change the code and avoid the previous coherence problem?

Problem 3 (2.5 points)
Given the following code:

```
#define SIZE_INDEX 256
#define N 1024*1024*1024

void histogram(unsigned int *S, int n, unsigned int *index) {
 unsigned int i, tmp;
 for (i=0; i<n; i++) {
   tmp = S[i]%SIZE_INDEX;
   index[tmp]++;
 } }

void main() {
  unsigned int S[N];
  unsigned int index[SIZE_INDEX];
  ...  // Here we have initialized S to random numbers and index to 0's
  histogram(S, N, index);
  ...
}
```

Write two different OpenMP parallel implementations of function `histogram` using the strategies presented below. You can modify the sequential code, add local variables and use any OpenMP pragma (except `omp for` worksharing-loop construct) and function you may need. Both implementations should minimize the use of synchronizations and load unbalance between threads during the processing of vector S.

1. (1 point) Cyclic Data Decomposition of the Output vector `index`.

2. (1.5 points) Block Data Decomposition of the Input vector `S`.

Problem 4 (2.5 points)

We ask you to write two additional parallel OpenMP implementations of the code that computes the `histogram`, this time using *task decomposition* strategies:

1. (1 point) Write an efficient OpenMP parallel version of the histogram program in Problem 3 using *explicit tasks*.

2. (1.5 points) Write a *recursive tree task decompostion* with *cutoff* based on the depth of the recursivity for the following code:

```
#define SIZE_INDEX 256
#define N 1024*1024*1024
#define BASE_SIZE 512
#define CUTOFF 3

void histogram(unsigned int *S, int n, unsigned int *index) {
 unsigned int i, tmp;
 for (i=0; i<n; i++) {
   tmp = S[i]%SIZE_INDEX;
   index[tmp]++;
} }

void histogram_rec(unsigned int *S, int n, unsigned int *index) {
  unsigned int n2=n/2;

  if (n > BASE_SIZE) {
    histogram_rec(S, n2, index);
    histogram_rec(&S[n2], n-n2, index);
  } else {
      histogram(S, n, index);
 }
}

void main() {
  unsigned int S[N];
  unsigned int index[SIZE_INDEX];
  ...
  // Here we have initialized S to random numbers and index to 0's
  histogram_rec(S, N, index);
  ...
}
```