PAR – Final Exam Theory/Problems– Course 2023/24-Q2

June 11^{th}, 2024

Problem 1 (2.5 points) Given the following C program instrumented with `Tareador`:

```
#define BS  4
#define N   16
int ii, jj, i, j;
char stringMessage[128];
int M[N][N];

for (ii=0; ii<N; ii+=BS)
  for (jj=0; jj<N; jj+=BS) {
    sprintf(stringMessage,"CoP(%d,%d)",ii,jj);
    tareador_start_task(stringMessage);

    for (i=ii; i<min(ii+BS,N) ; i++)
      for (j=max(1,jj); j<min(jj+BS,N-1) ; j++)
        M[i][j]= M[i][j] + M[i][j-1] + M[i][j+1] + color_pixel(i,j); // 10 t.u.

    tareador_end_task(stringMessage);
   }
```

Assume each iteration of the innermost loop body takes 10 time units, `BS` and `N` are 4 and 16, respectively, only matrix `M` is stored in memory (rest of variables are in registers), and function `color_pixel` only computes a value function of the induction variables `i` and `j`. **We ask you to:**

1. Draw the TDG of the parallel strategy above. Indicate which is the cost of each task.

   Solution:

   ```
   CoP(0,0)  -> CoP(0,4)  -> CoP(0,8)  -> CoP(0,12)
    (120)        (160)        (160)        (120)

   CoP(4,0)  -> CoP(4,4)  -> CoP(4,8)  -> CoP(4,12)
    (120)        (160)        (160)        (120)

   CoP(8,0)  -> CoP(8,4)  -> CoP(8,8)  -> CoP(8,12)
    (120)        (160)        (160)        (120)

   CoP(12,0) -> CoP(12,4) -> CoP(12,8) -> CoP(12,12)
    (120)        (160)        (160)        (120)
   ```

2. Compute T_1, T_∞ and P_{min}.

   Solution:

   $T_1 = 4(120 + 160 + 160 + 120)t.u. = 2240t.u.$
   $T_\infty = 120 + 160 + 160 + 120t.u. = 560t.u.$
   $P_{min} = 4$

3. Write the expression that determines the execution time T_4, clearly indicating the contribution of the computation time T_4^{comp} and data sharing overhead T_4^{mov}, for the two following assignments of tasks to processors:

#proc	Assignment 1 (task id)	Assignment 2 (task id)
0	CoP(0,0), CoP(0,4), CoP(0,8), CoP(0,12)	CoP(0,0), CoP(4,0), CoP(8,0), CoP(12,0)
1	CoP(4,0), CoP(4,4), CoP(4,8), CoP(4,12)	CoP(0,4), CoP(4,4), CoP(8,4), CoP(12,4)
2	CoP(8,0), CoP(8,4), CoP(8,8), CoP(8,12)	CoP(0,8), CoP(4,8), CoP(8,8), CoP(12,8)
3	CoP(12,0),CoP(12,4),CoP(12,8),CoP(12,12)	CoP(0,12),CoP(4,12),CoP(8,12),CoP(12,12)

You can assume: 1) a distributed-memory architecture with 4 processors; 2) matrix `M`, is initially distributed by rows in Assignment 1 (N/BS consecutive rows per processor) and initially distributed by columns in Assignment 2 (N/BS consecutive columns per processor); 3) once the loop is finished, you don't need the return matrix to their original distribution; 4) data sharing model with communication time per message $t_{comm} = t_s + m \times t_w$, being t_s, m, t_w the start–up time, the number of elements, and transfer time of one element, respectively; 5) BS perfectly divides N; and 6) the execution time for a single iteration of the innermost loop body takes 10 t.u..

Assignment 1 Solution:

$T_4 = T_4^{comp} + T_4^{mov}$
$T_4^{mov} = 0$; All data is local. No communication is needed.
$T_4^{comp} = (2 \times ((BS-1) \times BS) + (\frac{N}{BS} - 2) \times (BS \times BS)) \times 10t.u.$;

Assignment 2 Solution:

$P = 4$;
$T_4 = T_4^{comp} + T_4^{mov}$

$T_4^{mov} = t_s + N \times t_w + (P-2) \times (t_s + BS \times t_w) + \frac{N}{BS} \times (t_s + BS \times t_w)$;

$T_4^{comp} = ((BS-1) \times BS + (P-2) \times (BS \times BS) + (\frac{N}{BS} - 1) \times (BS \times BS) + (BS-1) \times BS) \times 10t.u.$;

Note : Regarding to the T_4^{mov}, there is a initial communication of the right boundary (first column of next processor), that all processor but last one have to do. This communication can be done by all processors in parallel. Then, each processor (but first one) hast to read BS elements of the left boundary, produced by a task in previous processor (dependence).

Regarding to T_4^{comp}, the last row of tasks executed by processor $P-1$ has $BS \times BS - 1$ elements to process each, but this processor has to wait for $\frac{N}{BS} - 1$ tasks of previous processor that last each for $BS \times BS \times 10t.u.$, therefore, this time should be taken into account for the first three tasks of last processor. Then, it has to process $BS \times BS - 1$ elements of the very last task.

Problem 2 (1.5 points) Consider the following sequential program that performs a matrix multiplication ($C = A \times B$):

```
int A[L][M], B[M][N], C[L][N];
int main() {
   int l, n, m;
   ...
   // Matrix Mutiplication
   for (l=0; l<L; l++)
      for (n=0; n<N; n++)
         for (m=0; m<M; m++)
             C[l][n] += A[l][m]*B[m][n];
   ...
   // Output results
   for (l=0; l<L; l++)
      for (n=0; n<N; n++)
          printf("C[%d][%d]=%d\n",l,n,C[l][n]);
}
```

We ask you to:

1. Write an OpenMP version to parallelize `Matrix multiplication` computation code using an iterative task decomposition strategy where: 1) the granularity of the tasks should be 2 iterations of the middle loop (loop n); and 2) the synchronization overheads are minimized.

   A Solution:

   ```
   ...
   int A[L][M], B[M][N], C[L][N];

   int main() {
      int l, n, m;
      ...

      #pragma omp parallel private(l,m)
      #pragma omp single
        for (l=0; l<L; l++)
           #pragma omp taskloop grainsize(2) nogroup
           for (n=0; n<N; n++)
              for (m=0; m<M; m++)
                 C[l][n] += A[l][m]*B[m][n];
      ...

      for (l=0; l<L; l++)
         for (n=0; n<N; n++)
             printf("C[%d][%d]=%d\n",l,n,C[l][n]);

   }
   ```

   Note: Tasks are independent among them. Thus, it is not necessary to add any synchronization between tasks neither for solving data race conditions. On the other hand, it is important to add nogroup to avoid taskgroup sinchronization, that is not necessary and we want to minimize synchronization overheads.

2. Do you think parallelizing the `Output results` code can improve the performance of the program while obtaining the expected output? Justify briefly your answer.

A Solution: The output results code have to executed sequentially as the order of executing between the different `printf`'s have to be preserved. For this reason, there is no point on parallelizing it as it will not improve performance.

Problem 3 (1.5 points) Consider the following recursive program that copies an array into another one:

```
double X[N],Y[N];

int copy(double * __restrict__ input, double * __restrict__ output, int n) {
   if (n<=32)
      for (int i=0; i<n; i++) output[i] = input[i];
   else {
      copy(input, output, n/2);
      copy(input+n/2, output+n/2, n-n/2);
} }

int main() {
   ...
   copy(X,Y,N);
   ...
}
```

We ask you to: write a parallel OpenMP program that performs a parallel and efficient recursive task decomposition, reducing the task creation overheads and minimizing data sharing synchronizations. Note that `__restrict__` indicates that input and output are not overlapped in memory.

A Solution:

```
SOLUTION:
...
#define CUTOFF 5

double X[N],Y[N];

int copy(double *__restrict__ input, double * __restrict__ output, int n, int depth) {
   if (n<32)
       for (int i=0; i<n; i++) output[i] = input[i];
   else {
    if (!omp_in_final()) {
      #pragma omp task final(depth>=CUTOFF)
      copy(input, output, n/2, depth+1);
      #pragma omp task final(depth>=CUTOFF)
      copy(input+n/2, output+n/2, n-n/2, depth+1);
    } else {
      copy(input, output, n/2, depth+1);
      copy(input+n/2, output+n/2, n-n/2, depth+1);
    }
  }
}

int main() {
        #pragma omp parallel
        #pragma omp single
        copy(X,Y,N,0);
}
```

Problem 4 (4.5 points) Given the following C code excerpt:

```
#define MAXHISTO P // P is the number of cores
typedef struct {
    double total;
    unsigned long long num;
} telem;
telem histo[MAXHISTO];
double v[MAXELEM]; // MAXELEM is a very large value
int i, pos;
 ...
// histo initialization phase
for (i=0; i<MAXHISTO; i++) {
   histo[i].total = 0;
   histo[i].num = 0;
}
// histo computation phase
for (i=0; i<MAXELEM; i++) {
   pos = getpos (v[i], MAXHISTO);   // returns a value between 0 and MAXHISTO-1
   // complex update of field "total" from vector "histo" at position "pos"
   complex_update (&histo[pos].total, v[i]);
   histo[pos].num ++;
}
```

We ask you to:

1. (1.5 points) Add the necessary `OpenMP` pragmas and clauses to parallelize the code in *histo computation* phase, on `P` cores, making use of *implicit tasks* and applying an *INPUT geometric block data decomposition*. The value `MAXELEM` is not necessarily a multiple of `P`. Your solution should maximize parallelism among implicit tasks.

   Solution:

```
omp_lock_t locks[P];
int i, pos;
 ...
// histo initialization phase
for (i=0; i<MAXHISTO; i++) {
   histo[i].total = 0;
   histo[i].num = 0;
   omp_init_lock (&locks[i]);
}
// histo computation phase
#pragma omp parallel private (i, pos) num_threads(P)
{
 int id = omp_get_thread_num();
 int BS = MAXELEM / P;
 int mod = MAXELEM % P;
 int start = id * BS;
 int end = start + BS;
 if (mod > 0) {
    if (id < mod) {
       start += id;
       end = start + BS + 1;
    } else {
       start += mod;
       end += mod;
    }
 }
 for (i=start; i<end; i++) {
```

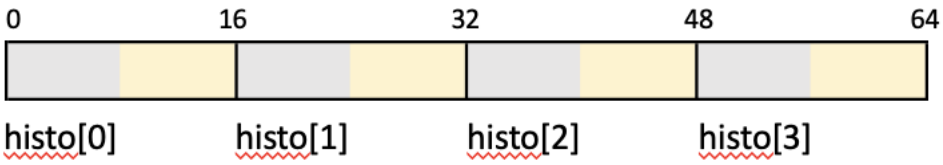
```
    pos = getpos (v[i], MAXHISTO);

    omp_set_lock (&locks[pos]);
    complex_update (&histo[pos].total, v[i]); // complex update of field "total" from vector
    omp_unset_lock (&locks[pos]);

    #pragma omp atomic
    histo [pos].num ++;
  }
 }
 ...
```

2. (1.5 points) Let's consider a shared-memory multiprocessor system composed by two NUMA nodes, each with two processors (cores) and 16 GBytes of shared main memory (MM). Each core has a private cache of 8MBytes. Cache and memory lines are 32 bytes wide. Cores 0 and 1 are allocated in NUMA node 0, and cores 2 and 3 are allocated in NUMA node 1. Data coherence is maintained using Write-Invalidate MSI protocols, with a Snoopy attached to each cache memory to provide coherency within each NUMA node and Write-Invalidate MSU directory-based coherence among the two NUMA nodes.

   (a) Consider `P=4` and draw a picture to show how many memory lines are necessary to allocate vector `histo` in the MM of the previously described system. You need to know that `sizeof(double) = 8 Bytes` and `sizeof(unsigned long long) = 8 Bytes`, the allocation of vectors `v` and `histo` are aligned to the start of memory line.

   Solution:

   To store vector `histo` it is necessary two memory lines (64 bytes):



   (b) **Compute the amount of bits taken by each snoopy** to maintain the coherence between caches inside a NUMA node and, **compute the amount of bits in each node directory** to maintain the coherence among NUMA nodes **ONLY for vector `histo`**.

   Solution:

   To store vector `histo` we need 2 memory lines. In order to keep coherency within a NUMA node using MSI Snoopy protocol, we need to keep 2 bits for each cache line. So the total number of bits needed by the MSI Snoopy protocol is **4 bits**.

   We need two entries in the directory structure to store the needed information to keep coherency. Each entry contain 2 bits for the state (MSU) and 2 bits for the sharers list. Consequently we need $2 * (2 + 2) =$ **8 bits** to keep coherency among NUMA nodes for vector `histo`.

   (c) Assuming that vector `histo` is entirely allocated in the MM of Numa Node 0, and all cache memories of the multiprocessor system are empty, **fill in the attached table** with the sequence of processor commands (Core), bus transactions within NUMA nodes (Snoopy), transactions yes/no between NUMA nodes (Directory), the presence bits, state for each cache and memory line, to keep cache coherence, AFTER the execution of each of the following sequence of commands:

   i. $core_2$ reads the contents of histo[1].total
   ii. $core_2$ writes the contents of histo[2].num
   iii. $core_1$ reads the contents of histo[0].total

3. (1.5 points) Let's assume that we want to execute the program on the shared-memory multiprocessor system described previously. Decide the most appropriate *geometric data decomposition* and write the parallel code in order to minimize synchronizations and reduce coherence traffic. You can also re-write the data structures.

Solution:

We define an OUTPUT geometric block data decomposition on vector `histo`, where the block size is equal to one element. In order to reduce coherence traffic generated by the false sharing when writing to different elements of `histo` but are allocated to the same cache line, we add padding (extra bytes) the the `telem` data structure. No synchronization is needed among implicit tasks as they update different elements of vector `histo`.

```
#define CACHE_LINE_SIZE 32
typedef struct {
    double total;
    unsigned long long num;
    char padding[CACHE_LINE_SIZE - sizeof(double) - sizeof(unsigned long long)];
} telem;
telem histo[MAXHISTO];
 ...
#pragma omp parallel
{
 int id = omp_get_thread_num();
 histo[id]=0;
 for (int i=0; i<MAXELEM; i++)
   int pos = getpos (v[i], MAXHISTO); // returns a value between 0 and MAXHISTO-1
   if (id == pos) {
      histo [pos].total += v[i];
      histo [pos].num ++;
   }
 }
 ...
```

Surname, name .....

Answers to **Question 4, Section 2.c**

Command	Coherence actions			Presence bits	State in MM	State in cache			
	Core	Snoopy	Directory (yes/no)			cache0	cache1	cache2	cache3
$core_2$ reads histo[1].total									
$core_2$ writes histo[2].num									
$core_1$ reads histo[0].total									

Solution:

Command	Coherence actions			Presence bits	State in MM	State in cache			
	Core	Snoopy	Directory (y/n)			cache0	cache1	cache2	cache3
$core_2$ reads histo[1].total	$PrRd_2$	$BusRd_2$	y	10	S	-	-	S	-
$core_2$ writes histo[2].num	$PrWr_2$	$BusRdX_2$	y	10	M	-	-	M	-
$core_1$ reads histo[0].total	$PrRd_1$	$BusRd_1$, $Flush_2$	n	11	S	-	S	S	-