PAR – In-Term Exam – Course 2024/25-Q1

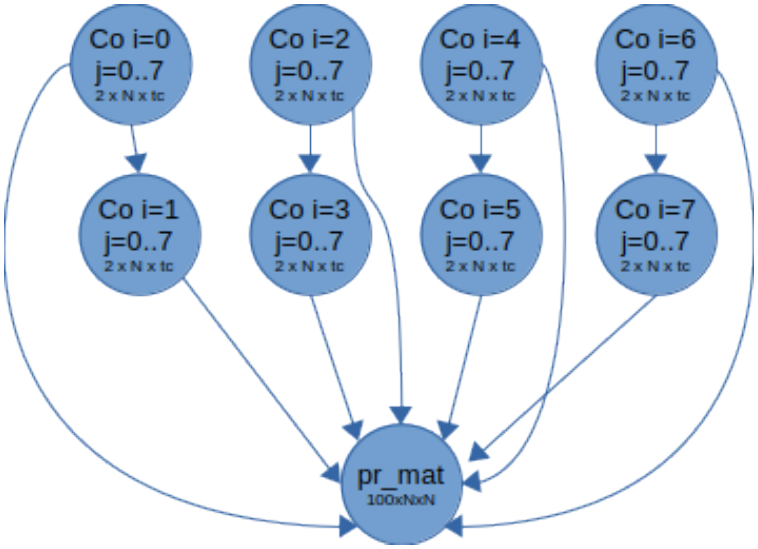
November 6^{th}, 2024

Problem 1 (4.0 points) Given the following code:

```
int mat[N][N];
void main() {
   for (int i=0; i<N; i++) {
     tareador_start_task("compute");
     for (int j=0; j<N; j++) {
        if (i%2 == 0)                          // cost of this line: tc t.u.
          mat[i][j] = mat[i][j] + mat[i+1][j]; // cost of this line: tc t.u.
        else
          mat[i][j] = mat[i][j] - mat[i-1][j]; // cost of this line: tc t.u.
     }
     tareador_end_task("compute");
   }
   tareador_start_task("print_matrix");
   print_matrix(mat);                          // cost of this function: 100*N*N t.u.
   tareador_end_task("print_matrix");
}
```

1. (1 point) Draw the Task Dependence Graph (TDG) based on the above Tareador task definitions and for `N=8`. Each task should be clearly labeled with the values of ranges of i, j, if needed, and its cost in time units (look at comments of the code to figure out the task cost lines).

   Solution:



   There are N `compute` tasks, one for each iteration of the outer loop `i`. The tasks computing even iterations have no Read After Write (RAW) dependencies. However, tasks computing odd iterations depend on the previous iteration. Each of these N `compute` tasks executes the whole set of iterations of the inner loop `j`, i.e. it performs the inner loop body N times. The associated cost of each task is $2 \times N \times t_c$ time units (which results in $16 \times t_c$ t.u. for $N = 8$) regardless of computing an even or odd iteration.

   A final task `print_matrix` prints the whole matrix. It depends on all the previous tasks since each row is previously updated by a different `compute` task. Its associated cost is $100 \times N^2$ time units.

2. (2.0 points) Assume `N=8` and the task costs of previous exercise. Compute the values for T_1, T_∞ and P_{min} and draw the temporal diagram for the execution of the TDG in the previous question on P_{min} processors.

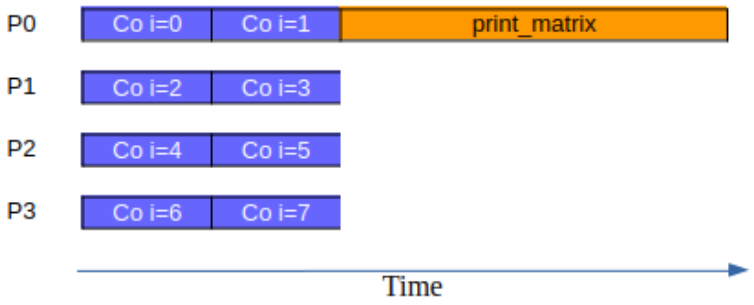
   Solution: Assuming `N=8`,

   $T_1 = N \times 2 \times N \times t_c + 100 \times N^2 = 128 \times t_c + 6400$

   $T_\infty = 2 \times 2 \times N \times t_c + 100 \times N^2 = \ \ 32 \times t_c + 6400$

   $P_{min} = 4$

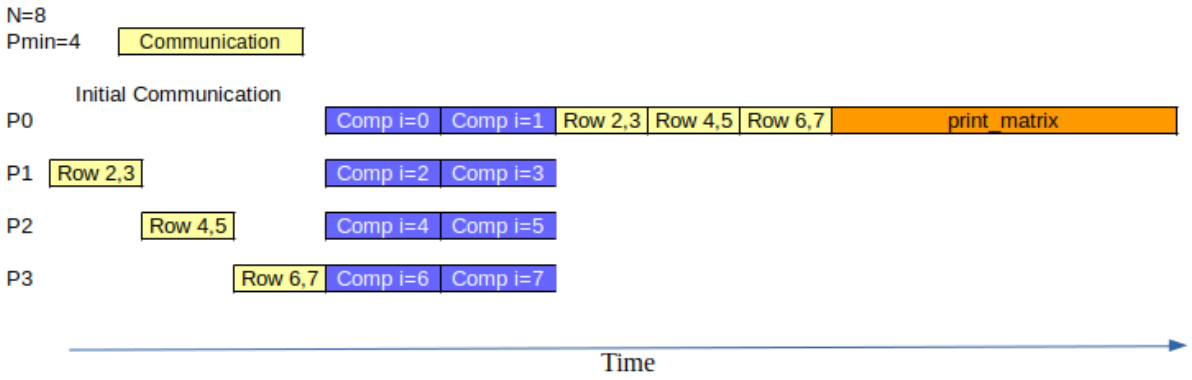
   Temporal diagram for the execution of the TDG in the previous question on $P_{min} = 4$ processors.



3. (1.0 points) Assume the same task definition, `N` a very large value multiple of 2, and consider a distributed memory architecture with $P = P_{min}$ processors. Let's assume that matrix `mat` is initially stored in the memory of processor 0 and tasks are scheduled to P processors in a similar way as you indicated in previous exercise, trying to minimize the data sharing overheads. Write the expression, function of N and $P = P_{min}$, that determines the execution time, T_p, clearly identifying the contribution of the computation time and the data sharing overheads, assuming the data sharing model explained in class in which the overhead to perform a remote memory 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 a given time, a processor can only perform one remote access to another processor and serve one remote access from another processor.

   Solution:

   Next, we show a temporal diagram for the execution of the tasks in the TDG shown in the 1^{st} question on $P_{min} = 4$ processors, considering the data sharing overheads. This is an example of the execution of the application for $N = 8$, for which $P_{min} = 4$. Note that processors start their computation once the initial communication is done, as explained in class, to symplify the data sharing model.



   In general, $P_{min} = \frac{N}{2}$, where each processor executes the set of two `compute` tasks which compute consecutive even-odd iterations. And one of them executes the final task `print_matrix`.

$T_{Computations} = 2 \times 2 \times N \times t_c + 100 \times N^2$

Prior to performing any computation, $P_1, P_2 \ldots P_{min} - 1$ need to receive 2 consecutive rows from P_0. Consequently, since P_0 can only serve one remote access at a time:

$T_{Initial\_Data\_Sharing} = (P_{min} - 1) \times (t_s + 2N \times t_w)$

Regardless of the processor in which task `print_matrix` is executed, it'll need to receive 2 consecutive rows from each of the other $P_{min} - 1$ processors. Thus,

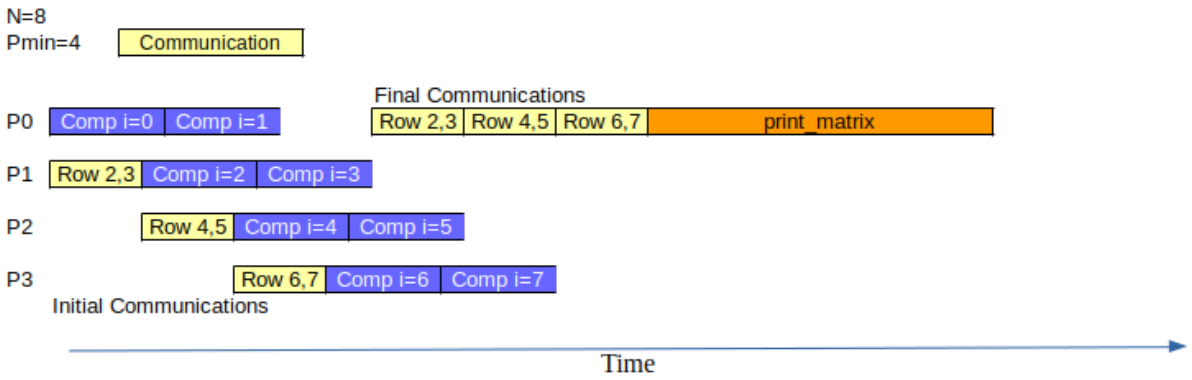
$T_{Final\_Data\_Sharing} = (P_{min} - 1) \times (t_s + 2N \times t_w)$

Therefore,

$$\begin{aligned} T_{P_{min}} &= T_{Initial\_Data\_Sharing} + T_{Computations} + T_{Final\_Data\_Sharing} \\ &= (P_{min} - 1) \times (t_s + 2N \times t_w) + 4 \times N \times t_c + 100 \times N^2 + (P_{min} - 1) \times (t_s + 2N \times t_w) \\ \\ &= 4 \times N \times t_c + 100 \times N^2 + 2 \times (P_{min} - 1) \times (t_s + 2N \times t_w) \\ &= 4 \times N \times t_c + 100 \times N^2 + 2 \times (\tfrac{N}{2} - 1) \times (t_s + 2N \times t_w) \\ &= 4 \times N \times t_c + 100 \times N^2 + (N - 2) \times (t_s + 2N \times t_w) \end{aligned}$$

Alternative optimized solution:

If we depart from the simplified model used in class and overlap communications with computations, then for $N = 8$:



And, in general:

$$\begin{aligned} T_{P_{min}} &= T_{Computations} + T_{Data\_Sharing} \\ &= 4 \times N \times t_c + 100 \times N^2 + (P_{min} - 1) \times (t_s + 2N \times t_w) + 1 \times (t_s + 2N \times t_w) \\ &= 4 \times N \times t_c + 100 \times N^2 + \tfrac{N}{2} \times (t_s + 2N \times t_w) \end{aligned}$$

Problem 2 (3.0 points) Consider the following sequential code:

```
int mat[N][N];

void main() {
    for (int i=0 ;i<N; i++)
      for (int j=0; j<N; j++)
        if (i%2 == 0)
             mat[i][j] = mat[i][j] + mat[i+1][j];
        else
             mat[i][j] = mat[i][j] - mat[i-1][j];
}
```

Assume N is a very large constant multiple of 2. Write an scalable OpenMP parallel program, being P the number of available processors, with the following conditions: 1) You are ONLY allowed to use explicit tasks, but NOT allowed to use taskloop; 2) The granularity of the task corresponds to the process of a chunk of BS consecutive rows of the matrix; 3) The synchronization needed to preserve the same result of the sequential execution, has to be minimized and 4) the overhead due to task creation and execution management has to be minimized.

Solution:

Analyzing the true dependencies in the sequential program we see that each odd row only depends on the previous (even) row. In this way, in order to minimize the synchronization we should guarantee that the chunk of rows assigned to a task should be an even number. In order to minimize the overhead of task management we could create a unique task for processing the maximum number of consecutive rows depending on N and P.

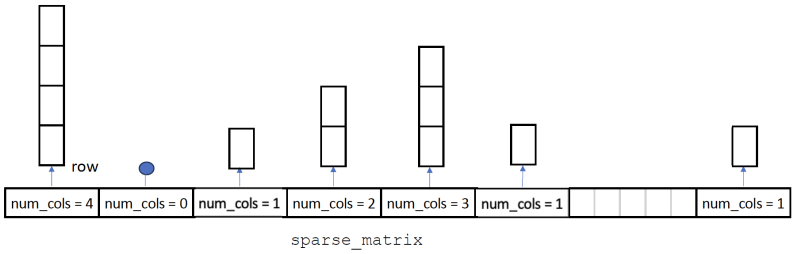
A possible solution:

```
int mat[N][N];

void main() {

 #pragma omp parallel num_threads(P)
 #pragma omp single
 {
    int BS = N/P;
    if (BS%2) BS++;
    for (int ii=0; ii<N; ii+=BS)
    #pragma omp task
    for (int i=ii; i<min(N,ii+BS); i++)
        for (int j=0; j<N; j++)
            if (i%2 == 0)
                mat[i][j] = mat[i][j] + mat[i+1][j];
            else
                mat[i][j] = mat[i][j] - mat[i-1][j];
 } }
```

We also could analyze the execution of each pair of even and odd rows and conclude that they could be processed jointly in the same iteration leading to the following code:

```
#pragma omp parallel num_threads(P)
#pragma omp single
{
    int BS = N/P/2;
    for (int ii=0; ii<N; ii+=BS*2)
    #pragma omp task
    for (i=ii; i<min(N,ii+BS*2); i+=2)
        for (j=0; j<N; j++) {
            int tmp = mat[i][j];
            mat[i][j] = mat[i][j] + mat[i+1][j];
            mat[i+1][j] = - tmp;
        }
```

Problem 3 (3.0 points) Given the following representation of a sparse matrix:



and the following data structures that model the sparse matrix with a sequential recursive algorithm to process it:

```
#define N ... // a large value and represents the number of rows
#define MIN_ROWS 2
#define MIN_NELEMS N
typedef {
    float value;
    int col;
} tRow;
typedef struct {
    int num_cols; // number of columns in the row
    tRow *row;
} tEntry;

// Perform calculation on first n rows of the sparse matrix.
float process (tEntry *sm, int n);

// Calculate pivot index (row index) such that the sum of num_cols (to process)
// up to row index (not included) is **similar** to the sum of num_cols (to process)
// at row index and after. This pivot index is returned in pointer int *index
// This total sum of num_cols before pivot index is also returned in *total_left
void partition (tEntry *sm, int n, int *index, int *total_left);

// Recursive calculation on the first n rows of sparse matrix sm
float rec_process (tEntry *sm, int n, int total_elems) {
    float result;
    if (n <= MIN_ROWS || total_elems <= MIN_NELEMS)
        result = process (sm, n);
    else {
        int index, total_left;
        partition (sm, n, &index, &total_left);
        result =  rec_process (sm, index, total_left);
        result += rec_process (&sm[index], n-index, total_elems-total_left);
    }
    return result;
}
int main() {
    tEntry sparse_matrix[N];
    int total_elems;
    ...
    float result = rec_process (sparse_matrix, N, total_elems);
    ...
}
```

We ask you to: Write a parallel OpenMP parallel version using a recursive task decomposition. Select the most appropriate strategy (tree or leaf) that will maximize the processor utilisation assuming a system with a high number of processors. The implementation should take into account the overhead due to task creation, limiting their creation and ensuring tasks are well-balanced.

Solution:

```
#define CUTOFF X // a value greater or equal than 2N
// Recursive calculation on the first n rows of sparse matrix sm
float rec_process (tEntry *sm, int n, int total_elems) {
    float result1, result2 = 0.0;
    if (n <= MIN_ROWS || total_elems <= MIN_NELEMS) {
        result1 = process (sm, n);
    }
    else {
        int index, total_left;
        partition (sm, n, &index, &total_left);
        if (!omp_in_final()) {
           #pragma omp task shared(result1) final (total_left < CUTOFF)
            result1 =  rec_process (sm, index, total_left);
           #pragma omp task shared(result2) final (total_elems-total_left < CUTOFF)
            result2 += rec_process (&sm[index], n-index, total_elems-total_left);
           #pragma omp taskwait
        }
        else {
            result1 =  rec_process (sm, index, total_left);
            result2 += rec_process (&sm[index], n-index, total_elems-total_left);
        }
    }
    return result1 + result2;
}
int main() {
    tEntry sparse_matrix[N];
    int total_elems;
    ...
    #pragma omp parallel
    #pragma omp single
    float result = rec_process (sparse_matrix, N, total_elems);
    ...
}
```

Notice that one task can achieve the cutoff condition, while the other one not, as the conditions are evaluated using different variables. This is in fact an advantage of marking a task as final individually, instead of performing a global condition.