PAR – Final Exam Theory/Problems– Course 2024/25-Q1

January 16th, 2025

Grade Publication January 22nd, 2025 - Exam Review 23rd - 16:00h–17:00h - C6E106

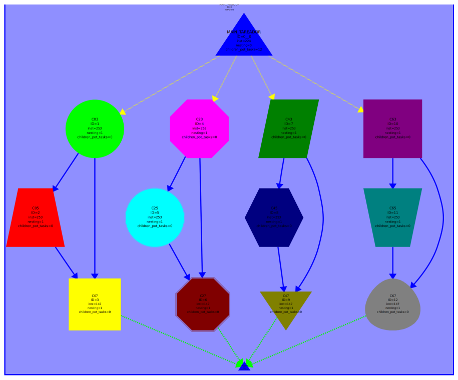
Problem 1 (2.5 points) Given the following C program instrumented with `Tareador`:

```
#define N 8
double M[N][N];

void main() {
char str[10];

  for (int ii=0; ii<N; ii+=2)
    for (int jj=3; jj<N; jj+=2) {
      sprintf(str,"C%d%d",ii,jj)
      tareador_start_task(str);
      for (int i=ii; i<ii+2; i++)
        for (int j=jj; j<min(jj+2,N); j++)
          M[i][j] = foo(M[i][j-1], M[i][j-2], M[i][j-3]);
      tareador_end_task(str);
    }
}
```

We ask you to:

1. Draw the TDG based on the above task definitions. Each task should be clearly labeled with the values for variables *ii* and *jj* at the time of task instantiation.

   Solution:


MAIN_TAREADOR
ID=0__0
inst=224
nesting=0
children_pot_tasks=12
C03 ID=1 inst=253 nesting=1 children_pot_tasks=0
C23 ID=4 inst=253 nesting=1 children_pot_tasks=0
C43 ID=7 inst=253 nesting=1 children_pot_tasks=0
C63 ID=10 inst=253 nesting=1 children_pot_tasks=0
C05 ID=2 inst=253 nesting=1 children_pot_tasks=0
C25 ID=5 inst=253 nesting=1 children_pot_tasks=0
C45 ID=8 inst=253 nesting=1 children_pot_tasks=0
C65 ID=11 inst=253 nesting=1 children_pot_tasks=0
C07 ID=3 inst=147 nesting=1 children_pot_tasks=0
C27 ID=6 inst=147 nesting=1 children_pot_tasks=0
C47 ID=9 inst=147 nesting=1 children_pot_tasks=0
C67 ID=12 inst=147 nesting=1 children_pot_tasks=0

2. Consider that the cost of executing function `foo` is constant and equal to 10 t.u. and the cost of the rest of the code is negligible. Calculate T_1, P_{min} and T_∞.

   Solution:

   T1 = 400 t.u. (40*4+40*4+20*4) Pmin = 4 Tinf = T4 = 100 t.u. (40+40+20)

3. Consider now that the program is executed on a distributed memory architecture with 4 processors and that matrix M is initially stored in the memory assigned to P_0. Consider the data sharing model explained in class in which the latency of a remote memory access is $t_s + t_w \times m$, where t_s is 5 t.u. and t_w is 1 t.u. Function `foo` takes 10 t.u. as stated in the last question. We ask you to calculate T_4 assuming the best task scheduling policy. There is no need to move data back to P_0 once memory is modified.

   Solution:

   T_4 = 100 + 3x16 t.u. = 148 t.u.

   Tasks $C_{ii,jj}$ are assigned to processor $P_{ii/2}$. All tasks executing in P_0 have the data they need. Tasks $C_{2,3}$, $C_{4,3}$ and $C_{6,3}$ need data from P_0, the first 3 elements in each row to be processed. 3 elements has to be read from P_0 from two different rows, and therefore, two messages of 3 elements are required: $2 \times (t_s + 3 \times t_w)$, 16 t.u. Data has to be read sequentially from processor 0, and the critical path will be stablished by the time to receive the data at the last processor reading from processor 0 plus the time to process the three tasks assigned to that processor.

Problem 2 (2.5 points) A *convolution* is an important operation in machine learning and signal processing. A *filter* is a small matrix storing values useful for detecting some pattern in the input. Through the convolution the filter slides across the input data (a larger matrix) in a step-wise manner. At each step, an *element-wise multiplication and addition* is performed: at each position, the filter's values are multiplied by the corresponding values in the input data under it. These products are then summed to produce a single output value. The process is repeated across the entire input, generating an output matrix that highlights the presence of the filter's specific feature. Consider the following sequential program that performs a *convolution* using a 3×3 filter. The input matrix has a halo (initial and final row and column initialized to zeros). Thus, the input matrix has two more rows and columns than the output matrix:

```
#define PADDED_N(N) ((N) + 2) // Dimension of padded input matrix size including halo
void applyConvolutionIterative(int *input, int *output, int *filter, int N);

int main() {
    int N = 1000;               // Output matrix dimension
    int paddedN = PADDED_N(N);  //  Input matrix dimension

    // Allocate and initialize input and output matrices
    int *output = (int *) calloc(N * N, sizeof(int)); // initializes to 0's
    int *input  = (int *) malloc(paddedN * paddedN * sizeof(int));
    init_input_matrix(input);

    // Example of a 3x3 filter
    int filter[3 * 3] = { 0, 1, 0,     1, -4, 1,     0, 1, 0 };

    applyConvolutionIterative(input, output, filter, N);
    ...
    print_matrix(output);
    free(input); free(output);
    return 0;
}

void applyConvolutionIterative(int *input, int *output, int *filter, int N) {
    int row, col, sum;
    for (row = 1; row <= N; row++) {
        for (col = 1; col <= N; col++) {
            sum = 0;

            // Apply the 3x3 filter to a neighborhood of 3x3 elements in the input matrix:
            for (int fi = -1; fi <= 1; fi++) {
                for (int fj = -1; fj <= 1; fj++) {
                    int ni = row + fi; // Neighbor row index
                    int nj = col + fj; // Neighbor column index

                    int neighbor_idx = ni * PADDED_N(N) + nj; // Row-wise storage
                    int filter_idx = (fi + 1) * 3 + (fj + 1);
                    // Perform the element-wise multiplication and addition:
                    sum += input[neighbor_idx] * filter[filter_idx];
                }
            }
            output[(row - 1) * N + (col - 1)] = sum; // Row-wise storage
        }
    }
}
```

Note: When writing your solutions you do NOT need to write the whole code. Instead, write the minimum code excerpts that allows the reader to understand clearly how the parallelization has been done.

We ask you to: write a parallel version of routine `applyConvolutionIterative` using OpenMP's explicit tasks. The solution must try to exploit the maximum parallelism among threads while minimizing the data sharing synchronization and task creation overheads.

Solution: There are no loop-carried dependencies in this code. In order to be as efficient as possible:
1) work must be split by rows since row-wise storage is used for both the input and the output matrix: and 2) tasks must have an appropriate granularity, namely a chunk of approximately $\frac{N}{NT}$ consecutive rows being NT the number of threads being used, departing from a very fine grain granularity because no load unbalancing is expected since all iterations should take the same amount of associated work.

One possible solution using `#pragma omp taskloop`:

```
void applyConvolutionIterative(int *input, int *output, int *filter, int N) {
    int row, col, sum;

    #pragma omp parallel
    #pragma omp single
    #pragma omp taskloop private(col, sum) grainsize(N / omp_get_num_threads())
    for (row = 1; row <= N; row++) {
        for (col = 1; col <= N; col++) {
            sum = 0;

            // Apply the 3x3 filter
            ... // Not changed

            output[(row - 1) * N + (col - 1)] = sum;
        }
    }
}
```

Another possible solution using `#pragma omp task`:

```
void applyConvolutionIterative(int *input, int *output, int *filter, int N) {
    int row, col, sum;
    #pragma omp parallel
    #pragma omp single
      {
        int BS = N / omp_get_num_threads();
        for (int row_start = 1; row_start <= N; row_start += BS) {
            int row_end = (row_start + BS - 1 <= N) ? row_start + BS - 1 : N;
            #pragma omp task  private(row, col, sum)  // default firstprivate(row_start, row_end)
            for (row = row_start; row <= row_end; row++) {
                for (col = 1; col <= N; col++) {
                    sum = 0;

                    // Apply the 3x3 filter
                    ... // Not changed

                    output[(row - 1) * N + (col - 1)] = sum;
                }
            }
        }
      }
}
```

We have a recursive implementation which we also want to parallelize.

```
#define PADDED_N(N) ((N) + 2) // Dimension of padded input matrix size including halo
#define THRESHOLD 50          // Base case threshold

void applyConvolutionBaseCase(int *input, int *output, int *filter, int N,
                              int row_start, int row_end);

void applyConvolutionRecursive(int *input, int *output, int *filter, int N,
                               int row_start, int row_end);
int main() {
    // Same main as in the previous question except for the call to the recursive version

    ...
    // Apply convolution recursively
    applyConvolutionRecursive(input, output, filter, N, 1, N);
    ...
}

// Iterative base case for convolution
void applyConvolutionBaseCase(int *input, int *output, int *filter, int N,
                              int row_start, int row_end) {
    for (int row = row_start; row <= row_end; row++) {
        for (int col = 1; col <= N; col++) {
            int sum = 0;

            // Apply the 3x3 filter
            ... // Same as in the iterative code in the previous question.

            output[(row - 1) * N + (col - 1)] = sum; // Row-wise storage
        }
} }

// Recursive function to apply convolution
void applyConvolutionRecursive(int *input, int *output, int *filter, int N,
                               int row_start, int row_end) {
    if (row_end - row_start + 1 <= THRESHOLD) {
        applyConvolutionBaseCase(input, output, filter, N, row_start, row_end);
        return;
    }

    int mid = (row_start + row_end) / 2;

    applyConvolutionRecursive(input, output, filter, N, row_start, mid);
    applyConvolutionRecursive(input, output, filter, N, mid + 1, row_end);
}
```

We ask you to: write a parallel version of routine `applyConvolutionRecursive` using OpenMP's explicit tasks. The solution must be efficient, exploiting parallelism as soon as possible but not on the leaves of the recursivity. In addition the code must be prepared to control the creation of tasks based on the depth of the recursivity and a value stored in a constant named MAX_DEPTH.

Solution: In order to exploit parallelism as soon as possible we have to implement a *Tree* decomposition. The solution implements *cutoff* based on the depth of the recursivity.

```
#define MAX_DEPTH 4            // Maximum depth for task creation

// Iterative base case for convolution is not changed at all
...

// Recursive function to apply convolution
void applyConvolutionRecursive(int *input, int *output, int *filter, int N,
                               int row_start, int row_end, int depth) {
    if (row_end - row_start + 1 <= THRESHOLD) {
        applyConvolutionBaseCase(input, output, filter, N, row_start, row_end);
        return;
    }

    int mid = (row_start + row_end) / 2;

    if ( !omp_in_final() ) {
        #pragma omp task final(depth >= MAX_DEPTH)
        applyConvolutionRecursive(input, output, filter, N, row_start, mid, depth + 1);

        #pragma omp task final(depth >= MAX_DEPTH)
        applyConvolutionRecursive(input, output, filter, N, mid + 1, row_end, depth + 1);

    } else {
        applyConvolutionRecursive(input, output, filter, N, row_start, mid, depth + 1);
        applyConvolutionRecursive(input, output, filter, N, mid + 1, row_end, depth + 1);
    }
}


int main() {
    // Same main as in the previous question except for the call to the recursive version

    ...
    // Apply convolution recursively in parallel
    #pragma omp parallel
    #pragma omp single
    applyConvolutionRecursive(input, output, filter, N, 1, N, 0);
    ...
}
```

Problem 3 (2.5 points) Given the following code fragment computing matrix `A[N][N]`, an `N` much larger than the number of processors `P` (particularly `N » 4P`).

```
#define N .. // a value much larger than P
#define k N/4

float A[N][N], B[N][N];
...
for (int i = k; i < N-k; i++)
    for (int j = i; j < N-k; j++)
         A[i][j] = foo (A[i][j-k], A[i][j], B[i-k][j]); // computation function
```

We ask you to: Decide the most appropriate *Input/Output Geometric Data Decomposition* for matrices `A` and `B` and write a parallel version of the code by adding the *OpenMP* necessary directives and clauses, making use of only implicit tasks. Your solution proposal should:

- Minimize the synchronization overhead among implicit tasks
- Minimize the load unbalance
- Maximize data locality

Solution:

The best option is a geometric cyclic data decomposition of matrix A and matrix B by rows along the P processors, starting allocation on processor 0, of row k for matrix A, and of row 0 (i-k) for matrix B. This will help to exploit data locality and avoid data dependence synchronization.

```
#define N .. // a value much larger than P
#define k N/4

float A[N][N], B[N][N];
...
#pragma omp parallel num_threads(P)
{
 int myid = omp_get_thread_num();

 for (int i = k+myid; i < N-k; i += P)
    for (int j = i; j < N-k; j++)
         A[i][j] = foo (A[i][j-k], A[i][j], B[i-k][j]);
}
```

Surname, name ....

Problem 4 (2.5 points)

Assume the definition and allocation of matrices of the previous exercise. In addition, consider a shared-memory multiprocessor system composed by two NUMA nodes, each with two processors (cores) and 64 GBytes of shared main memory (MM). Each core has a private cache of 32 MBytes. Memory lines are 64 bytes long (one memory line per memory page), the allocation in memory for matrix `A` and matrix `B` are aligned to the start of a memory line and `sizeof(float)` is 4 bytes.

Cores 0 and 1 belong to 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.

Assume now that matrix `A` 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:

Memory access	Core	Coherence commands		Presence bits	State in MM	State in cache			
		Snoopy	Directory (yes/no)			cache0	cache1	cache2	cache3
$core_0$ reads `A[0][0]`									
$core_3$ reads `A[0][5]`									
$core_1$ writes `A[0][15]`									
$core_2$ writes `A[1][15]`									

Solution:

Memory access	Core	Coherence commands		Presence bits	State in MM	State in cache			
		Snoopy	Directory (yes/no)			cache0	cache1	cache2	cache3
$core_0$ reads A[0][0]	$PrRd_0$	$BusRd_0$	*no*	01	S	S	-	-	-
$core_3$ reads A[0][5]	$PrRd_3$	$BusRd_3$	*yes*	11	S	S	-	-	S
$core_1$ writes A[0][15]	$PrWr_1$	$BusRdX_1$	*yes*	01	M	I	M	-	I
$core_2$ writes A[1][15]	$PrWr_2$	$BusRdX_2$	*yes*	10	M	-	-	M	-