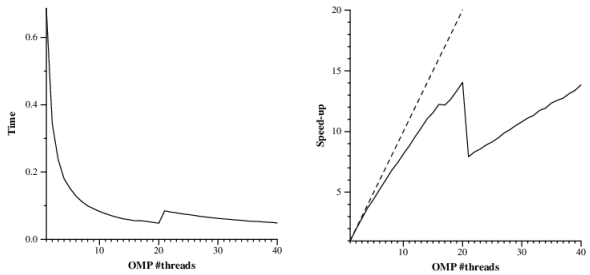
PAR – Final Exam Laboratory– Course 2023/24-Q1

January 17^{th}, 2023

Problem 1: Lab 1 (2.5 points)

Let's assume the `pi_omp` code in *Lab 1*. Below we include the execution time and speedup scalability plots obtained with the `submit-strong-omp.sh` script when setting `np_MAX=40`. Recall that this script executes the parallel code using from 1 to `np_MAX` threads. **We ask you to:** Briefly explain the reason why the speed-up goes down abruptly when going from 20 to 21 *OpenMP* threads, and why the performance for 20 and 40 *OpenMP* threads is very similar.



Solution: No solution provided. Look at the Atenea survey of Questions laboratory 1 - Session 1 - third question.

Problem 2: Lab 3 (2.5 points)

1. Assume the following task decomposition strategies for the `dot_product` code:

```
// Strategy 1
float result = 0.0;
#pragma omp parallel
#pragma omp single
#pragma omp taskloop reduction(+:result)
for (int i = 0; i < n; ++i) {
   result += x[i] + y[i];
}
```

```
// Strategy 2
float result = 0.0;
#pragma omp parallel
#pragma omp single
#pragma omp taskgroup task_reduction(+: result)
for (int i = 0; i < n; ++i) {
   #pragma omp task in_reduction (+:result)
   result += x[i] + y[i];
}
```

   We ask you to: Indicate which of the two strategies would obtain better scalability. Justify briefly your answer.

   Solution:

   Strategy 2 applies a task decomposition with task granularity much finer than Strategy 1. Observe that in Strategy 2, tasks are created for every loop iteration, while in Strategy 1, tasks are created with a bunch of consecutive iterations (taskloop behaviour). The work performed at each loop iteration (accumulation on a variable with a sum up) do not justify the task creation at this level. We can affirm because in Laboratory 3, where for Mandelbrot set calculation (which involved a greater number of operations), the Point strategy suffered from a great task creation overhead, and Row strategy showed better performance.

2. Take a look at the two versions of *Modelfactors* tables generated after executing the *Mandelbrot* application parallelized using the *Point* strategy and `taskloop` pragma to create the explicit tasks.

   We ask you to: Analyze the main differences between the two executions and indicate which is the optimization applied to *Parallelization 2*. Reason briefly your answer.

   Solution: No solution provided. Look at your laboratory deliverable and feedback

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	3200.0	12800.0	25600.0	38400.0	51200.0
LB (number of explicit tasks executed)	1.0	0.93	0.84	0.53	0.48
LB (time executing explicit tasks)	1.0	0.95	0.95	0.95	0.96
Time per explicit task (average us)	129.09	33.54	18.3	12.66	9.68
Overhead per explicit task (synch %)	0.04	26.9	102.7	229.43	456.08
Overhead per explicit task (sched %)	0.33	3.84	12.88	20.52	30.87
Number of taskwait/taskgroup (total)	320.0	320.0	320.0	320.0	320.0

Figure 1: Modelfactors table for Parallelization 1

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	3200.0	12800.0	25600.0	38400.0	51200.0
LB (number of explicit tasks executed)	1.0	0.51	0.62	0.75	0.93
LB (time executing explicit tasks)	1.0	0.98	0.98	0.97	0.97
Time per explicit task (average us)	129.1	33.49	18.22	12.69	9.71
Overhead per explicit task (synch %)	0.0	3.6	64.34	180.14	396.75
Overhead per explicit task (sched %)	0.31	2.29	10.68	18.35	28.53
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0

Figure 2: Modelfactors table for Parallelization 2

Problem 3: Lab 4 (2.5 points)

Consider the following sequential recursive version for the `dot_product` problem presented in the assignment for *Lab 4*:

```
#define N 512
#define MIN_SIZE 64

int iter_dot_product(int *A, int *B, int n);

int rec_dot_product(int *A, int *B, int n) {
   int res1, res2=0;
   if (n>MIN_SIZE) {
       int n2 = n / 2;
       res1 = rec_dot_product(A, B, n2);
       res2 = rec_dot_product(A+n2, B+n2, n-n2);
   }
   else res1 = iter_dot_product(A, B, n);
   return res1+res2;
}

void main() {
   int result;
   result = rec_dot_product(a, b, N);
}
```

We ask you to:

1. Assuming we only add the necessary OpenMP directives to parallelize the previous code following a Recursive Task Decomposition using a *Leaf* strategy, indicate which would be the **Maximum** *Instantaneous Parallelism* we can achieve. Reason your answer. **Note:** Including the parallel code in your answer is not mandatory.

   Solution: The maximum instantaneous parallelism that we can achieve is 1. Observe that the `rec_dot_product` function cannot return until `res1` and `res2` are calculated. So in the case of a leaf strategy, as tasks are created sequentially, and have to finish execution before returning the result, there won't be more than one task executing at any given moment.

2. Assuming we only add the necessary OpenMP directives to parallelize the previous code following a Recursive Task Decomposition using a *Tree* strategy, indicate which would be the **Maximum** *Instantaneous Parallelism* we can achieve. Reason your answer. **Note:** Including the parallel code in your answer is not mandatory.

   Solution: The maximum instantaneous parallelism that we can achieve is 8, after three levels of recursive calls and one task created per recursive call (two tasks per level). Note that after three recursive levels the base case of the recursivity is achieved, when 8 tasks can be potentially executed in parallel.

Problem 4: Lab 5 (2.5 points)

Assume a parallelization strategy for the Gauss-Seidel solver in *Lab5* that uses a *geometric block by rows data decomposition*. Remember that the code in the laboratory assignment included an argument, *userparam*, to be used to determine the number of blocks each thread has to process.

1. Indicate why *userparam* impacts directly on the amount of parallelism obtained. Reason your answer.

   Solution: No solution provided. Look at your laboratory deliverable and feedback.

2. Explain the reason why in *Lab5* assignment you needed to implement your own synchronization between threads.

   Solution:

   When working with implicit tasks, there is no way to specify dependencies using OpenMP clauses. Consequently, all the implicit tasks will be executed simultaneously without taking care of dependencies between blocks.