PAR – In-Term Exam – Course 2023/24-Q2

April 4^{th}, 2024

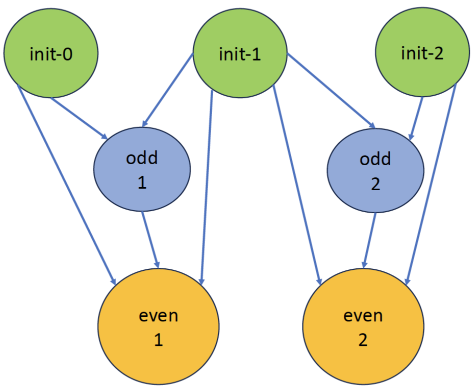
Problem 1 (4.0 points) Given the following code:

```
#define N  ... // value determined at each section
float A[N][N], B[N][N];
...
// initialization
 for (int i=0; i<N; i++) {
   tareador_start_task("init");
   for (k=0; k<N; k++) {
        A[i][k] = init(); // 10 time units
   }
   tareador_end_task("init");
}
// calculation
 for (int i=1; i<N; i++) {
   tareador_start_task("odd"); // Process only odd columns
   for (k=1; k<N; k+=2) {
        B[i][k] = A[i-1][k] + A[i][k] + foo(); // 20 time units
   }
   tareador_end_task("odd");
   tareador_start_task("even"); // Process only even columns
   for (k=2; k<N; k+=2) {
        B[i][k] = A[i-1][k] + A[i][k] + B[i][k-1] + goo(); // 40 time units
   }
   tareador_end_task("even");
 }
```

Assuming that functions `foo`, `goo` and `init` do not modify any element from matrix A and matrix B, **we ask you to:**

1. (1.0 points) Draw the Task Dependence Graph (TDG) based on the Tareador task definitions in the instrumented code above. Each task should be clearly labeled with the value of i and its cost in time units. Assume that $N = 3$.

   Solution:



   Where the cost for each `init` task is 30 time units, for each `odd` task is 20 time units and for each `even` task is 40 time units.

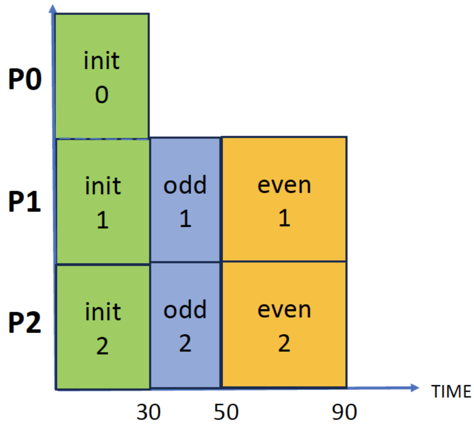
2. (1.0 points) Compute the values for T_1, T_∞ and P_{min}. Draw the temporal diagram for the execution of the TDG if executed using P_{min} processors.

Solution:

$T_1 = 30 \times 3 + (20 + 40) \times 2 = 210$ time units

$T_\infty = 30 + 20 + 40 = 90$ time units

$P_{min} = 3$

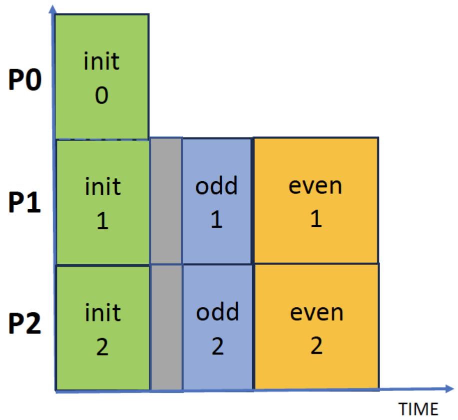


3. (2.0 points) Assume the same task definition, and consider a distributed memory architecture with P processors. Let's assume that matrix A and matrix B are distributed by rows along the P processors (row `i` on processor `i`). Tasks are scheduled on the processor where is allocated the data they have to update. This means that a task that works on row `i` executes on processor `i`.

We ask you to:

(a) (1.0 points) Draw the time diagram for the execution of the tasks in P processors clearly identifying the computation and the data sharing time. Note: to answer this question you can assume $N = 3$.

Solution:



The grey bars correspond to the data sharing time.

(b) (1.0 points) Write the expression that determines the execution time T_p as a function of N and P. Identify clearly 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. Remember that at a given time, a processor can only perform one remote access to another processor and serve one remote access from another processor.

Solution:

$T_P = T_{comp} + T_{comm}$

It is necessary to distinguish if N is an odd or an even value:

i. If N is an odd value, the expression for T_{comp} is:
$T_{comp} = 10 \times N + (20 \times (N-1)/2 + 40 \times (N-1)/2)$

ii. If N is an even value, the expression for T_{comp} is:
$T_{comp} = 10 \times N + (20 \times N/2 + 40 \times (N-2)/2)$

$T_{comm} = ts + (N-1) \times t_w$

Every task `odd-i` and `even-i` needs 2 rows from matrix `A`. One of the rows is available in the local processor and the other has to be transferred from processor `i-1`, which is the one that counts for the data sharing overhead.

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

```
object_type *mat[256][256];
int histo[5] = {0, 0, 0, 0, 0};
int object_val(object_type *p); // return value of object *p in the range 0..4

int main() {
     for (int i=0 ; i<256; i++)
        for (int j=0; j<256; j++)
            histo[object_val(mat[i][j])]++;
}
```

that computes the histogram of the values in the range [0..4] related to the objects pointed by a square matrix of 256x256 elements.

We ask you to: Write an iterative task decomposition strategy using OpenMP that efficiently exploits the parallellism reducing task creation and synchronization overheads. You can propose a solution based on implicit or explicit tasks.

Solution with implicit tasks:

```
int main() {
   #pragma omp parallel
   {
    int BS = 256/omp_get_num_threads(); // assume nt divides 256
    int id = omp_get_thread_num();
    int histo_l[5] = {0,0,0,0,0};
    for (int i=id*BS ; i<(id+1)*BS; i++)
       for (int j=0; j<256; j++)
           histo_l[object_val(mat[i][j])]++;

    for (int i=0; i<5; i++)
       #pragma omp atomic
       histo[i] += histo_l[i];
   }
}
```

Solution with explicit tasks:

```
int main() {
   #pragma omp parallel
   #pragma omp single
   {
    int BS = 256/omp_get_num_threads();
    for (int ii=0; ii<256; ii+=BS)
    {
     #pragma omp task
     {
       int histo_l[5] = {0,0,0,0,0};
       for (int i=ii ; i<ii+BS; i++)
         for (int j=0; j<256; j++)
             histo_l[object_val(mat[i][j])]++;

       for (int i=0; i<5; i++)
          #pragma omp atomic
```

```
            histo[i] += histo_l[i];
        }

      }

    }
}
```

Solution with explicit tasks - taskloop:

```
int main() {
   #pragma omp parallel
   #pragma omp single
   {
    #pragma omp taskloop num_tasks(omp_get_num_threads())
    for (int i=0 ; i<256; i++)
      for (int j=0; j<256; j++)
      {
          int tmp=object_val(mat[i][j]));
          #pragma omp atomic
          histo[tmp]++;
      }
   }
}
```

Problem 3 (3.0 points) Given the following sequential code:

```
unsigned int fibbonacci(unsigned int n) {
    unsigned int fib;
    if(n == 0)       fib = 0;
    else if(n == 1)  fib = 1;
    else {
       fib = fibbonacci(n-1);
       fib += fibbonacci(n-2);
    }
    return fib;
}
int main() {
   unsigned int fibnumber;
   ...
   fibnumber=fibbonacci(N)
   printf("Fibbonacci(%d)\n",fibnumber);
   ...
}
```

We ask you to:

1. (1.5 points) Create a parallel version in OpenMP using a recursive task decomposition for the `fibbonacci` function. Select the most appropriate strategy (*tree* or *leaf*) that will maximize the processor utilisation assuming a system with a high number of processors and without considering task creation and synchronization overheads.
2. (1.0 points) Modify the previous code to implement a task generation control mechanism based on the depth level. Use `MAX_DEPTH` as the maximum depth level to decide if tasks must be created or not.

3. (0.5 points) Assume your first parallel version. Which type of synchronization is required to guarantee that your tasks have finished before the `printf` statement in the main program if you can not assume any implicit or explicit thread barrier between the fibbonacci and `printf` calls?.

Solution a+b+c:

The correct parallel strategy is the tree recursive task decomposition.

The code already modified with the cutoff mechanism follows:

```
unsigned int fibbonacci(unsigned int n, int depth) {
    unsigned int fib1, fib2, fib;
    if(n == 0){
       fib = 0;
    } else if(n == 1) {
       fib = 1;
    } else {
      if (!omp_in_final())
      {
       #pragma omp task shared(fib1) final(depth>=MAX_DEPTH)
       fib1 = fibbonacci(n-1, depth+1);
       #pragma omp task shared(fib2) final(depth>=MAX_DEPTH)
       fib2 = fibbonacci(n-2, depth+1);
       #pragma omp taskwait
      }
      else
      {
       fib1 = fibbonacci(n-1, depth+1);
       fib2 = fibbonacci(n-2, depth+1);
      }
      fib = fib1+fib2;
    }
    return fib;
}

int main() {
   unsigned int fibnumber;
   ...

   #pragma omp parallel
   #pragma omp single
   {
      fibnumber=fibbonacci(N,0)
      // 3) nothing is needed because
      //    each parallel level has a taskwait synch
      printf("Fibbonacci(%d)\n",fibnumber);
   }
   ..
}
```

We do not need any taskwait neither taskgroup since each parallel level already waits for its tasks.