# Parallel and Distributed Computing

2020/2021

1st Exercise Class

1. Consider that, after porting an application to a multiprocessor machine, it executes in three modes:

   A - all processors are used

   B - only half of the processors are used

   C - serial execution (no paralellism)

   Assume that, in a single processor machine, the execution spends 0.02% of the time in the serial mode (C) part of the application.

   Consider also that the multiprocessor has 100 processors.

   Determine the maximum fraction of time, relative to the execution time in the single processor machine, that the application may spend in mode B such that the performance of the multiprocessor is 80 faster than that of the monoprocessor.

2. Consider an application executing in a multiprocessor with 32 processors, running at 1GHz. Access time to a remote memory is 400ns, during which the processor is blocked until the request is completed. Assuming that, on average, each processor is able to execute 2 instructions per clock cycle, how much faster is the execution of an application that does not require remote accesses compared to an application where 0.2% of the instructions cause a remote access?

3. The following program adds two square matrices `a` and `b`:

```
#define SIZE ... /* Size of the matrices */
int i, j;        /* 32 bit integers */
int a[SIZE,SIZE], b[SIZE,SIZE], s[SIZE,SIZE];

for(i = 0; i < SIZE; i = i+1)
    for(j = 0; j < SIZE; j = j+1)
        s[i,j] = a[i,j] + b[i,j];
```

   A concurrent version of this program was written for a multiprocessor with distributed shared memory, where the sum is split over $N$ processes each executing on a different processor.

   The primitive `parallel` spawns the concurrent execution of the subsequent statements until the `join` primitive. The program blocks at the `join` waiting for all concurrent

statements to finish. The source program is pre-processed before actual compilation so that the number of statements between `parallel` and `join` is evenly distributed among $N$ processors.

```
#define N ... /* Number of processors */
#define SIZE ...
int a[SIZE,SIZE], b[SIZE,SIZE], s[SIZE,SIZE];

partial_sum(int i_min, int iter) /* Adds a subset of lines of the matrices */
{
  int i,j;

  for(i = i_min; i < i_min+iter; i = i+1)
      for(j = 0; j < SIZE; j = j+1)
          s[i,j] = a[i,j] + b[i,j];
}

main()
{
  parallel; /* Spawns N partials sums */
    partial_sum(0, SIZE/N);
    partial_sum(SIZE/N, SIZE/N);
    partial_sum(2*SIZE/N, SIZE/N);
    partial_sum(3*SIZE/N, SIZE/N);
    ...
    partial_sum ((N-1)*SIZE/N, SIZE/N);
  join;
}
```

The following execution times were measured in the multiprocessor:

| Function | Execution Time |
|---|---|
| Routine `partial_sum()` | 100ns/iteration |
| Start-up/finish<br>remote process | $1\mu$s<br>($1\mu$s = 500ns at local node + 500ns at remote node ) |
| Data transfer | 1ns/byte |

The data transfer time refers to the time required for a process to load in its address space its assigned range of input data, and, in the opposite direction, the time for a process to send the computed data to the main processor. Assume an asynchronous operation, where the data transfer occurs in the background while the process continues its execution.

a) For this program, determine the *speedup*, $S$, as a function of the number of processors, $N$, and the dimension of the matrices, $SIZE$: $S(N, SIZE)$

b) Compute the *speedup* for $N = 100$ processors, and for matrices with sizes $100 \times 100$ and $10000 \times 10000$. Discuss the results obtained.