Goal

Implement **task parallelism** in the provided code using OpenMP's **parallel**, **sections**, and **barrier** directives to enhance performance. Key Analysis Points: (1) **Load Balancing**: Explicitly investigate the impact of task load balancing. If threads have unequal workloads, propose and test a strategy to distribute the work more evenly between threads. (2) **Scheduling Strategy**: Analyze how the internal task scheduling strategy (e.g., how OpenMP assigns section blocks to threads) influences performance.

Code Example

The program on the right executes five functions sequentially, repeating this process REP times. The implementation of each function is shown below.

```
double  v = 10.0;
for ( i=0; i<REP; i++ )
{
  VectF1      (A, B, N);
  VectF2      (B, C, v, N);
  VectScan    (C, A, N);
  VectAverage (B, D, N);
  v = VectSum (D, N);
}
```

Below are the compilation and profiling commands using the default input arguments. The **–fno-inline** flag instructs the compiler to disable function inlining, ensuring the profiler can accurately measure performance for each individual function.

```
$ gcc -Ofast -fno-inline Prg.c -o Prg
$ perf stat Prg
$ perf record Prg
$ perf report
```

```
void VectF1 ( double *IN, double *OUT, int n)
{
  for ( int i=0; i<n; i++ ) {
    long int T = IN[i];
    OUT[i] = (double) (T % 4) + 0.5 + (IN[i]-trunc(IN[i]));
  }
}

void VectF2 ( double *IN, double *OUT, double v, int n)
{
  for ( int i=0; i<n; i++ )
    OUT[i] = v / ( 1.0 + fabs(IN[i]));
}

void VectScan ( double *IN, double *OUT, int n)
{
  double sum = 0.0;
  for ( int i=0; i<n; i++ )
  {
    sum += IN[i];
    OUT[i] = sum; // Inclusive: include current element
  }
}

void VectAverage ( double *IN, double *OUT, int n)
{
  for ( int i=1; i<n-1; i++ ) {
    OUT[i] = (2.0*IN[i]+IN[i-1]+IN[i+1])/4.0;
  }
}

double VectSum (double *V, int n)
{
  double sum = 0;
  for ( int i=0; i< n; i++ )
    sum = sum + V[i];
  return sum;
}
```

P1. Compile the previous program and profile the execution.

The program execution time is approximately 12.3 seconds. Using the profiling data from the provided screen capture, estimate the total time taken by each of the five functions and record your results in the table. Please note the output values and be mindful of how they change when the program is modified.

Time (s)	F1 (s)	F2 (s)	Scan (s)	Average (s)	Sum (s)
12.3					
Inputs: N= 20000, Rep= 250000					
Outputs: v= 5.012831130447e+04, A[19999]= 3.237046462115e+08					

```
44,72%  Prg    Prg            [.] VectF1
18,55%  Prg    Prg            [.] VectF2
18,43%  Prg    Prg            [.] VectScan
 9,33%  Prg    Prg            [.] VectSum
 8,96%  Prg    Prg            [.] VectAverage
 0,00%  Prg    libc-2.28.so   [.] intel_check_word.isra.0
 0,00%  Prg    ld-2.28.so     [.] _dl_sysdep_start
 0,00%  Prg    ld-2.28.so     [.] _dl_start
 0,00%  Prg    ld-2.28.so     [.] _start
```

P2. Analyze, compile and run the provided parallel code using **parallel**, **single**, **sections/section**, and **barrier** directives

The main loop of the provided program has been modified with OpenMP directives to parallelize the execution of several functions (see code below). You must perform the following tasks:

- **Compile and Execute**: Before running the program, ensure you define the environment variable that sets the number of OpenMP threads. Then, compile and run the program as instructed.
- **Analyze Output**: Carefully observe the program's output.
- **Dependency Graph**: Based on your analysis of the code and output, draw a task dependency graph. This graph should clearly show the execution order constraints enforced by the program. Use arrows to indicate dependencies (e.g., an arrow from Function A to Function B means A must complete before B can begin). You must understand the effect of the **omp** directives on the execution of the program.
- **Correctness**: What issues are apparent in the program's output (i.e., the final calculated values)? Explain why these incorrect results occur. Compare the Dependency Graph of this parallel program with the serial execution of the previous section.

```
$ gcc -Ofast -fno-inline -fopenmp PrgPAR.c -o PrgPAR
$ export OMP_NUM_THREADS=2
$ PrgPAR
```

```
#pragma omp parallel shared (A,B,C,D,v)
{
  #pragma omp single
  VectF1 (A, B, N);

  #pragma omp sections
  {
    #pragma omp section
    VectF2      (B, C, v, N);
    #pragma omp section
    VectScan    (C, A, N);
  }
  VectAverage (B, D, N);
  #pragma omp barrier
  v = VectSum (D, N);
}
```

```
Inputs: N= 20000, Rep= 2
Thread 1 starts executing F1
Thread 1 ends executing F1
Thread 0 starts executing F2
Thread 1 starts executing Scan
Thread 1 ends executing Scan
Thread 0 ends executing F2
Thread 1 starts executing Average
Thread 0 starts executing Average
Thread 0 ends executing Average
Thread 1 ends executing Average
Thread 0 starts executing Sum
Thread 1 starts executing Sum
Thread 0 ends executing Sum
Thread 1 ends executing Sum
Thread 1 starts executing F1
Thread 1 ends executing F1
Thread 0 starts executing F2
Thread 1 starts executing Scan
Thread 0 ends executing F2
Thread 1 ends executing Scan
Thread 1 starts executing Average
Thread 0 starts executing Average
Thread 1 ends executing Average
Thread 0 ends executing Average
Thread 1 starts executing Sum
Thread 0 starts executing Sum
Thread 1 ends executing Sum
Thread 0 ends executing Sum
Outputs: v= 5.523626674810e+04, A[19999]= 5.399171295359e+07
```

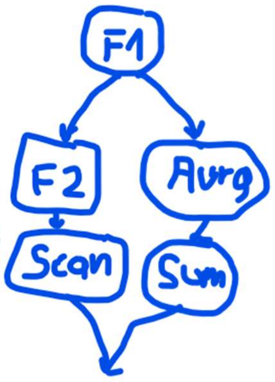
P3. Analyze, compile, run and profile the correct parallel code (**parallel**, **single**, **sections/section**, and **barrier** directives)

A correct parallel version of the code is provided below. Tasks: (1) **Correctness Analysis**: Explain the design choices in this version to ensure its correctness. Specifically, describe how it correctly manages data dependencies and synchronization to produce the right result. (2) **Performance Prediction**: Using two threads for the parallel region, predict the expected execution time. Provide a brief justification for your prediction (e.g., by identifying the parallelizable sections and estimating the potential speedup). (3) **Empirical Validation**: Measure the actual execution time of the program using two threads. Compare this empirical result with your prediction and discuss any discrepancies.

```
$ gcc -Ofast -fno-inline -fopenmp PrgPAR2.c -o PrgPAR2
$ export OMP_NUM_THREADS=2
$ perf stat PrgPAR2
```

```
#pragma omp parallel shared (A,B,C,D,v)
{
  #pragma omp single
  VectF1 (A, B, N);

  #pragma omp sections
  {
    #pragma omp section
    {
      VectF2      (B, C, v, N);
      VectScan    (C, A, N);
    }
    #pragma omp section
    {
      VectAverage (B, D, N);
      v = VectSum (D, N);
    }
  }
}
```



P4. Program optimization for best performance when running 2 threads

Optimize the program to achieve the fastest possible execution time using two threads. Document and justify all changes. The parallel version of the program must produce functionally identical results to the original. Adopt an incremental development methodology: make small, controlled changes and verify the correctness of the program after each one. This practice ensures robust and reliable software development.

Hint: A primary optimization is to parallelize function **VectF1**. The provided code fragment demonstrates how to partition its task into two equal halves (assuming **N** is even) for parallel execution. Use this as a starting point for further optimizations.

```
#pragma omp sections
{
  #pragma omp section
  VectF1      (A, B, N/2);

  #pragma omp section
  VectF1      (A+N/2, B+N/2, N/2);
}
```

P5. Program optimization for best performance when running 6 or 12 threads (the processor contains 6 execution cores)

Optimize the program to achieve a faster execution time using multi-threading. The theoretical maximum speedup is challenging to achieve, with a 6× improvement being particularly difficult. In your report, you must: Document your optimization attempts, including any that did not improve performance; Analyze the bottlenecks that ultimately prevented a near-linear speedup; Explain the fundamental difficulties, such as sequential sections, synchronization overhead, or memory contention, that limited your performance gains.