Task 0.1

```
#include <omp.h>
#define N 1000
#define CHUNKSIZE 100

main(int argc, char *argv[]) {

    int i, chunk;
    float a[N], b[N], c[N];

    /* We initialize the vectors */
    for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;

    #pragma omp parallel shared(a,b,c,chunk) private(i)
    {
  #pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
  c[i] = a[i] + b[i];
    } /* end of parallel region */
}
```

Task 0.1.1

In the above code snippet, the ‘chunk’ variable is used to specify the chunk size the parallel scheduler uses. It indicates how the scheduler should distribute the loop iterations among the available threads. The chunk size 100 means each available thread gets assigned 100 iterations of the parallel ‘for’ loop.

Task 0.1.2

“#pragma omp parallel shared(a,b,c,chunk) private(i)”
The above line of code starts a parallel region where a team of threads is created, and the enclosed code block below it will be executed in parallel by these threads.

“shared(a, b, c, chunk)” indicates that the arrays a, b, and c, as well as the variable chunk, are shared among all threads. This means that each thread has access to these variables, and any modification made by one thread is visible to all other threads.

“private(i)” indicates that each thread has its private copy of the loop index i. This is necessary because each thread will have its own copy of the loop index to avoid data races and ensure that each thread works with its own distinct data.

Task 0.1.3

“schedule(dynamic, chunk)” distributes the loop iterations between available threads and specifies a dynamic scheduling policy with a chunk size of ‘chunk’, which is 100. With dynamic scheduling, iterations are dynamically assigned to threads as they become available, meaning that if certain loops take more time, it won’t cause a bottleneck.

Other options other than “dynamic” include “auto”, “guided”, “runtime”, and “static”.

Task 0.1.4

The main metric used is execution time and the resulting speed-up. We can also measure the number of loop iterations and corresponding execution times, as well as overhead time and load balancing.

Task 1: Study of the OpenMP API

Introduction:

OpenMP is an application programming interface (API) that provides a high-level interface to simplify parallel programming. It is a set of compiler directives and library routines for parallel programming in C, C++, and Fortran.

Compilation:

We will focus on OpenMP's implementation using c/c++ and therefore our compiler will be g++. The code should contain the line *#include <omp.h>*, and the flag "-fopenmp" needs to be added in the compilation command.

Directive:

The standard directive looks like this:

```
#pragma omp parallel
{
        // some code to be executed in parallel…
}
```

The directive 'parallel' indicates that the following code block needs to be initialized for parallelisation. Other different directives can be used to specify how the code should be paralellised. These directives include:

'for': loop iterations are shared between different threats depending on further parameters.

'critical': indicates a section of critical code that can only be executed by one thread at a time.

'section': allows user to indicate different sections of code to be executed in parallel.