

## Module 9 — November 6, 2025

### Solutions

#### I. Using the OpenMP wallclock timer:

```
#include <omp.h>
double t1, t2;
double tomp1;
...
t1 = omp_get_wtime();
for (i = 0; i < N; i++)
    omp_dgemv_v1(1.0, 0.0, A, x, y);
t2 = omp_get_wtime();
tomp1 = (t2 - t1) / N;
...
```

`omp_get_wtime()` returns the time in seconds! If you want to have the time in ms, like Module 7, you have to multiply the result by 1000.

Another solution is, to use a similar macro, as shown in Module 7's solution:

```
#ifdef _OPENMP
#include <omp.h>
#define mytimer omp_get_wtime
#define delta_t(a,b) (1e3 * ((b)-(a)))
#else
#include <time.h>
#define mytimer clock
#define delta_t(a,b) (1e3 * ((b) - (a)) / CLOCKS_PER_SEC)
#endif
```

and somewhere inside `main()` :

```
#ifdef _OPENMP
    double t1, t2;
    fprintf(stderr, "OpenMP version: timing wallclock time (in ms)! ");
#else
    clock_t t1, t1;
    fprintf(stderr, "Serial version: timing CPU time (in ms)!\n");
#endif
```

Note: `omp_get_wtime()` returns the result in `double`, while `clock()` returns the result as `clock_t` !

**II.** Using `init_data()` and `check_results()` :

```

#include "datatools.h"

...
/* Allocate memory */
order = RowMajor;
A = array2d_alloc((size_t[]){m,n}, order);
x = array_alloc(n);
y = array_alloc(m);
r = array_alloc(m);
if (A == NULL | x == NULL | y == NULL | r == NULL) {
    fprintf(stderr, "Memory allocation error!\n");
    exit(EXIT_FAILURE);
}

...
/* initialize with useful data - last argument is reference */
init_data(m,n,y,A,x,r);

...
/* check the results - bail out if an error is encountered */
if (check_results("row", m, n, y, r) > 0) exit(EXIT_FAILURE);

```

**III.** Implement the generic function `my_dgemv()` :

```

int
my_dgemv(double alpha,
         double beta,
         const array2d_t * A,
         const array_t * x,
         array_t * y) {

    if (!A || !x || !y)
        return 1;
    if (A->shape[1] != x->len || A->shape[0] != y->len)
        return 1;

    size_t m = A->shape[0];
    size_t n = A->shape[1];
    double *px = x->val;
    double *py = y->val;

    if (A->order == RowMajor) {
        for (size_t i = 0; i < m; i++) {
            py[i] *= beta;
            double *pA = A->val + i * n;
            double sum = 0.0;

```

```

        for (size_t j = 0; j < n; j++) {
            sum += alpha * pA[j] * px[j];    // stride 1
        }
        py[i] += sum;
    }
}
else {
    for (size_t i = 0; i < m; i++)
        py[i] *= beta;

    for (size_t j = 0; j < n; j++) {
        double *pA = A->val + j * m;

        for (size_t i = 0; i < m; i++) {
            py[i] += alpha * pA[i] * px[j];    // stride 1
        }
    }
}
return 0;
}

```

#### IV. Implement the function `omp_dgemv()` :

```

int
omp_dgemv(double alpha,
          double beta,
          const array2d_t * A,
          const array_t * x,
          array_t * y) {

    if (!A || !x || !y)
        return 1;
    if (A->shape[1] != x->len || A->shape[0] != y->len)
        return 1;

    size_t m = A->shape[0];
    size_t n = A->shape[1];
    double *px = x->val;
    double *py = y->val;

    if (A->order == RowMajor) {
        #pragma omp parallel for
        for (size_t i = 0; i < m; i++) {
            py[i] *= beta;
            double *pA = A->val + i * n;
            double sum = 0.0;

```

```
        for (size_t j = 0; j < n; j++) {
            sum += alpha * pA[j] * px[j];    // stride 1
        }
        py[i] += sum;
    }
}
else {
    for (size_t i = 0; i < m; i++)
        py[i] *= beta;
    #pragma omp parallel
    {
        for (size_t j = 0; j < n; j++) {
            double *pA = A->val + j * m;

            #pragma omp for
            for (size_t i = 0; i < m; i++) {
                py[i] += alpha * pA[i] * px[j];    // stride 1
            }
        }
    }
}

return 0;
}
```

Timings (obtained on a 16-core Xeon E5-2665, Scientific Linux 7.7, GCC 9.2.0):

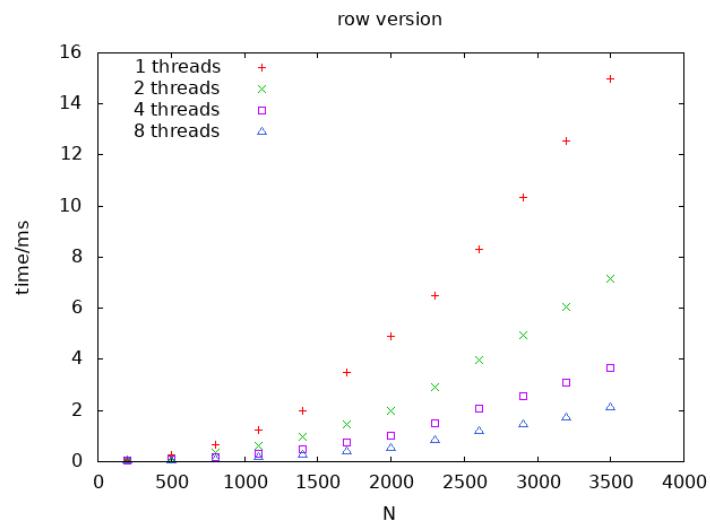


Figure 1: Row version: timings

As we can see, the runtime is reduced when adding more threads - but the effects gets smaller and smaller for larger number of threads. Thus, it is better to look at the speed-up.

Speed-up values:

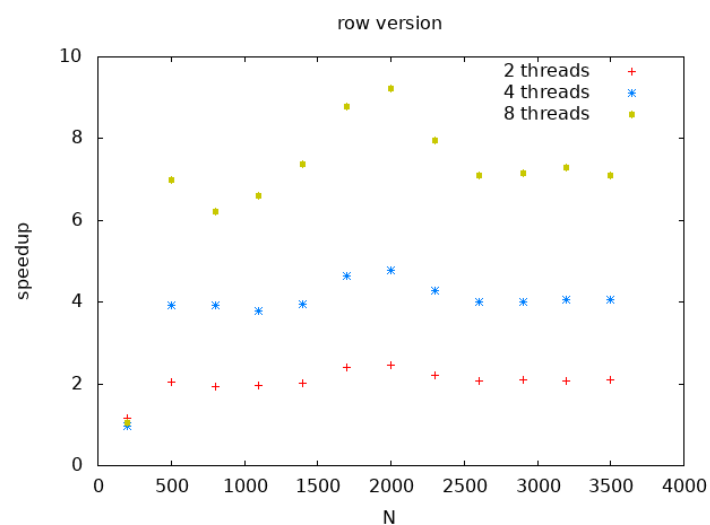


Figure 2: Row version: speed-up values

We get good speed-up values for 2, 4 and 8 threads, except for the smallest matrix sizes. In some range, we can even observe speed-ups above the theoretical value, which is a cache effect. The drop in speed-up for 8 threads and larger matrices is related to non-optimal memory access!

Timings for the column-wise version (same system as above):

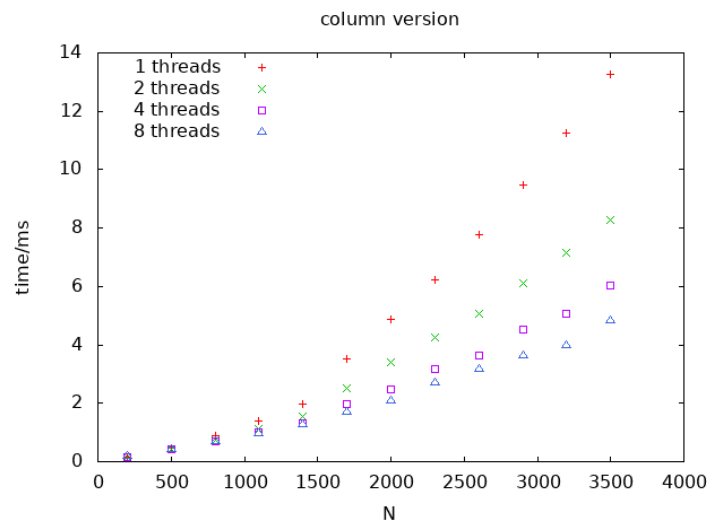


Figure 3: Column version: timings

The timing curves look quite good, as we get lower runtimes when using more cores. Now let's have a look at the speed-up values for the column-wise version:

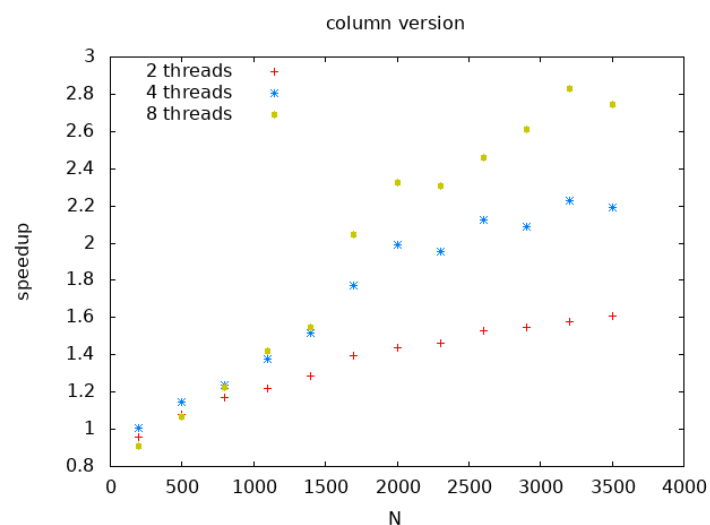


Figure 4: Column version: speed-up values

The speed-up values here are not as good as for the row-wise version above, although we used column-wise access for column major storage! This is a consequence of the way the work inside the parallel region is distributed, and how memory is accessed. There are ways to “fix” this problem, but this is beyond the scope of this course.