02635 Fall 2016 — Module 6 (solutions)

Exercises

I. Timing `datasize1()`:

```
#include <stdio.h>
#include <stdlib.h>

// for the timings
#include <time.h>
#define mytimer clock
#define delta_t(a,b) (1.0e3 * ((b) - (a)) / CLOCKS_PER_SEC)
#define MIN_RTIME 2000    // run iterations for at least MIN_RTIME msecs

// the external function, and some sizes
extern int datasize1(int);
#define MAX_SIZE 16777216 // 128*1024*1024/8 elements

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

    clock_t t1, t2;
    double tcpu;
    int mem_acc;
    int iter;

    printf("# Testing function datasize1:\n");
    for(int i = 2048; i <= MAX_SIZE; i *= 2) {
        tcpu = 0.0; iter = 0;
        t1 = mytimer();
        do {
            mem_acc = datasize1(i);
            t2 = mytimer();
            tcpu = delta_t(t1, t2);
            iter++;
        } while (tcpu < MIN_RTIME);
        printf("%.2lf %.2lf\n", (double)i*8/1024, // memory in kB
                                tcpu / iter );    // time per iter in ms
    }
    return(0);
}
```

To get the performance in Mflop/s, replace the `printf(...)` statement in the code above by

```
printf("%.2lf %.2lf\n", (double)i*8/1024,               // memory in kB
                        (1.0e-3*iter)*mem_acc / tcpu ); // Mflop/s
```

II.

1. Implement the function `my_dgemv_v1`:

```
void my_dgemv_v1(
  int m,          /* number of rows                   */
  int n,          /* number of columns                */
  double alpha,   /* scalar                           */
  double ** A,    /* two-dim. array A of size m-by-n  */
  double * x,     /* one-dim. array x of length n     */
  double beta,    /* scalar                           */
  double * y      /* one-dim. array x of length m     */
) {
  int i,j;
  for (i=0;i<m;i++) {
    y[i] *= beta;
    for (j=0;j<n;j++) {
      y[i] += alpha*A[i][j]*x[j];
    }
  }
  return;
}
```

2. See exercise 5 below.

3. Implement the function `my_dgemv_v2`:

```
void my_dgemv_v2(
  /* ... lines not shown here .../
) {
  int i,j;
  for (i=0;i<m;i++) y[i] *= beta;
  for (j=0;j<n;j++) {
    for (i=0;i<m;i++) {
      y[i] += alpha*A[i][j]*x[j];
    }
  }
  return;
}
```

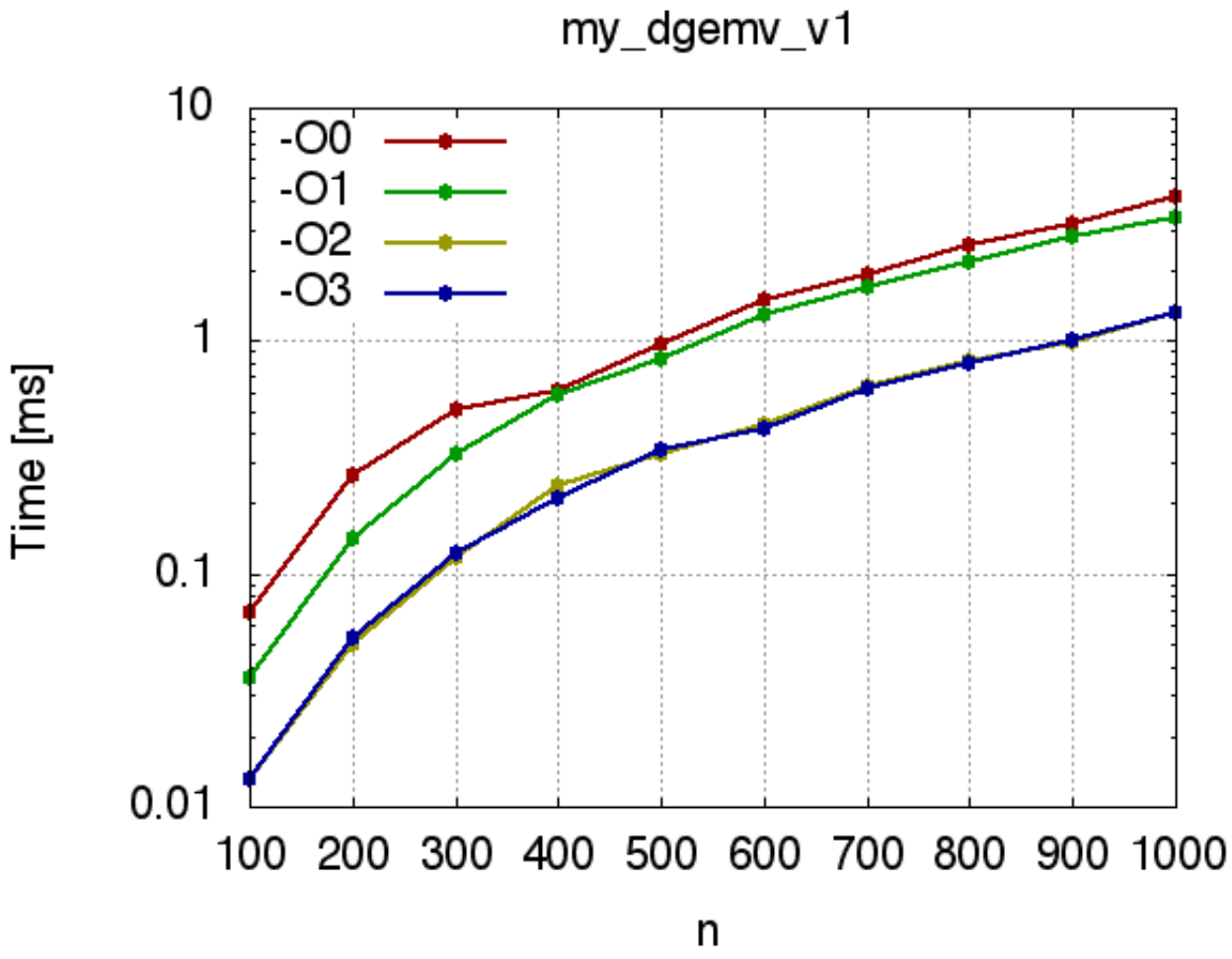
4. See exercise 5 below.

5. Repeat the two timing experiments with compiler optimizations:

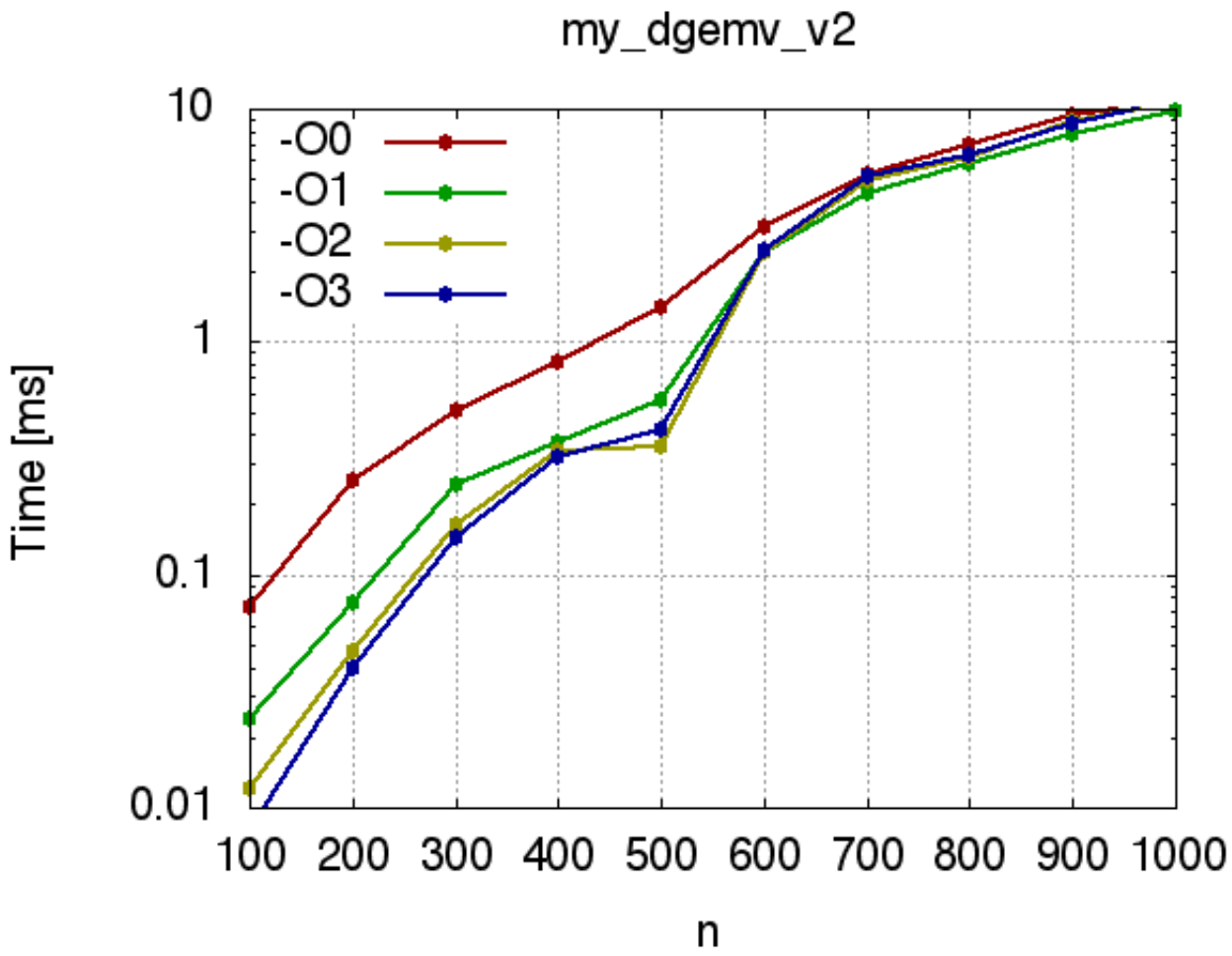
```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define NREPEAT 100

/* ################################################ */
/* ### Insert my_dgemv_v1 and my_dgemv_v2 here ### */
/* ################################################ */

/* Routine for allocating two-dimentional array */
double ** malloc_2d(int m, int n) {
  int i;
  if (m <= 0 || n <= 0) return NULL;
  double ** A = malloc(m*sizeof(double *));
  if ( A == NULL ) return NULL;
  A[0] = malloc(m*n*sizeof(double));
  if (A[0] == NULL) {free(A); return NULL;}
  for (i=1;i<m;i++) A[i] = A[0] + i*n;
  return A;
}

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

  int i,m,n,N = NREPEAT;
  double *x, *y, **A, tcpu1, tcpu2;
  clock_t t1,t2;

  for (m=100;m<=1000;m+=100) {
    n = m;

    /* Allocate memory */
    A = malloc_2d(m,n);
    x = malloc(n*sizeof(*x));
    y = malloc(m*sizeof(*y));
    if ( A == NULL || x == NULL | y == NULL ) {
       fprintf(stderr, "Memory allocation error...\n");
       exit(EXIT_FAILURE);
    }

    /* CPU time for my_dgemv_v1 */
    t1 = clock();
    for (i=0;i<N;i++)
       my_dgemv_v1(m,n,1.0,A,x,0.0,y);
    t2 = clock();
    tcpu1 = 1e3*(t2-t1)/CLOCKS_PER_SEC/N;
```

```
      /* CPU time for my_dgemv_v2 */
      t1 = clock();
      for (i=0;i<N;i++)
         my_dgemv_v2(m,n,1.0,A,x,0.0,y);
      t2 = clock();
      tcpu2 = 1e3*(t2-t1)/CLOCKS_PER_SEC/N;

      /* Print n and results */
      printf("%4d %8.3f %8.3f\n",n,tcpu1,tcpu2);

      /* Free memory */
      free(A[0]);
      free(A);
      free(x);
      free(y);
    }
    return EXIT_SUCCESS;
  }
```

CPU time required by `my_dgemv_v1()` :



CPU time required by `my_dgemv_v2()`:



The results show that the first implementation, `my_dgemv_v1`, benefits quite a bit from compiler optimization (for all n). Moreover, it is significantly faster than the second version (`my_dgemv_v2`), especially for large n. Indeed, the spacial locality is much better in the first variant of method since it accesses the elements of A row-by-row in accordance with the row-major storage.

Note that the CPU times may differ (significantly) on other systems.