Computação de Alto Desempenho

UFRJ 2011/1

Comparando multiplicação de matrix por vetor em C e Fortran

Rodolfo Henrique Carvalho

A proposta do trabalho é comparar o tempo de execução de 4 rotinas que computem:

$v \leftarrow Ax$

Tal que A é uma matriz N x N, e x é um vetor N x 1. Consequentemente, v é N x 1.
As 4 rotinas computam a multiplicação da seguinte maneira:

1. Acesso por linha e depois por coluna em C;
2. Acesso por coluna e depois por linha em C;
3. Acesso por linha e depois por coluna em Fortran;
4. Acesso por coluna e depois por linha em Fortran;

A complexidade computacional do problema é $O(n^2)$, mas como será o comportamento real observado? Qual o impacto da linguagem? Qual o impacto de percorrer linhas-colunas ou colunas-linhas?

Usando os códigos em anexo, foram geradas as seguintes tabelas que mostram o tempo de execução do produto de matrix por vetor em cada um dos 4 casos:

C

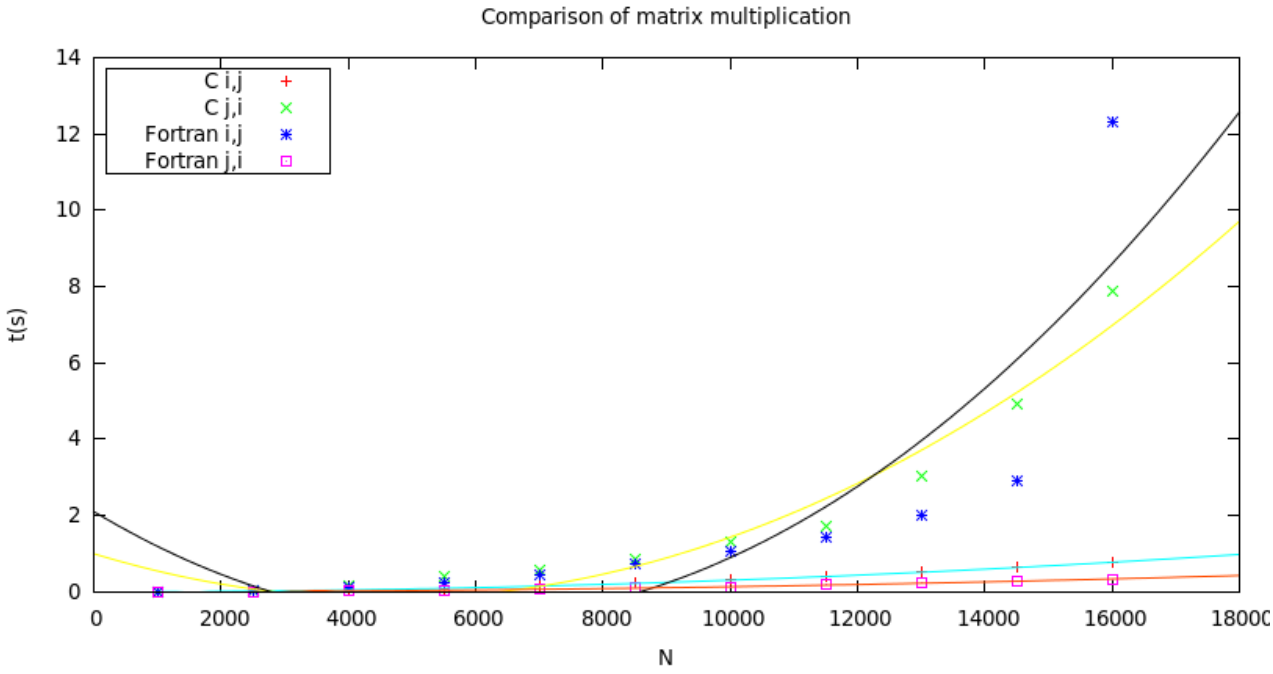
```
[1] Compute v = Ax looping through rows then columns (i, j)
[2] Compute v = Ax looping through columns then rows (j, i)
     N                [1]            [2]
  1000      0.000000      0.010000
  2500      0.020000      0.050000
  4000      0.050000      0.170000
  5500      0.090000      0.410000
  7000      0.150000      0.580000
  8500      0.220000      0.850000
 10000      0.310000      1.310000
 11500      0.400000      1.720000
 13000      0.510000      3.040000
 14500      0.630000      4.910000
 16000      0.780000      7.870000
```

Fortran

```
[1] Compute v = Ax looping through rows then columns (i, j)
[2] Compute v = Ax looping through columns then rows (j, i)
     N                [1]            [2]
  1000      0.010000      0.000000
  2500      0.040000      0.010000
  4000      0.120000      0.020000
  5500      0.250000      0.040000
  7000      0.450000      0.060000
  8500      0.710000      0.100000
 10000      1.059999      0.130000
 11500      1.420000      0.179999
 13000      1.990001      0.230000
 14500      2.900000      0.270000
```

```
16000       12.299999       0.340000
```

A seguir, a mesma informação plotada em gráficos com respectivas regressões polinomiais do tipo ax^2+bx+c:



Nota-se que as regressões para C j,i e Fortran i,j claramente não são adequadas, enquanto que as outras duas comportam-se conforme o esperado em $O(n^2)$.

Em C o desempenho é melhor quando lemos linhas-colunas, e em Fortran é o contrário, colunas-linhas. Isso se deve pela forma como cada uma das linguagens aloca e estrutura posições de memória para um *array*.

Em C, a matrix fica armazenada por linhas, de forma que quando lemos por linha, o elemento da próxima coluna vai estar no cache e teremos uma alta taxa de hit. Quando acessamos por colunas, estamos indo contra a arquitetura, e acessar uma nova posição (i, j) acarreta um miss no cache e consequente maior tempo de execução.

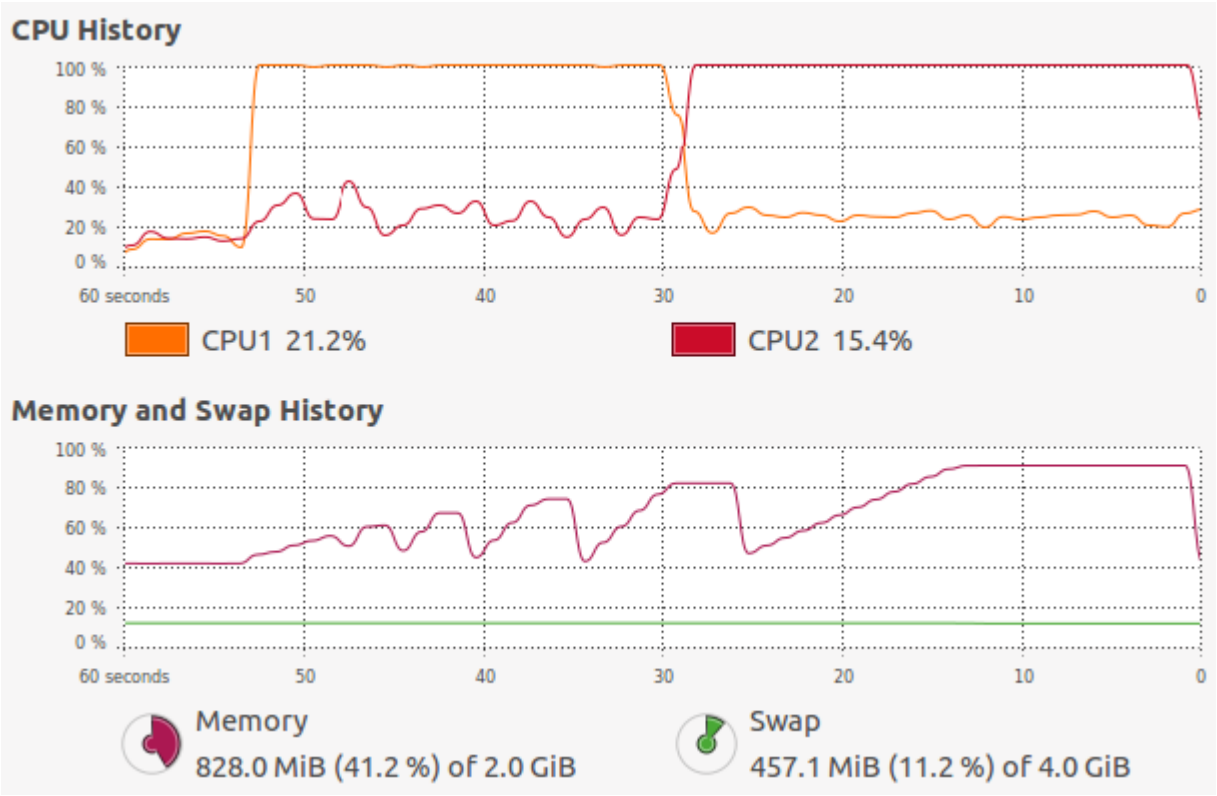
Já em Fortran, o resultado foi o oposto pois as matrizes são armazenadas por coluna. Neste caso, ao acessar uma posição (i, j), “toda” a coluna j vai estar em cache e acessar i+1, i+2, etc, é rápido.

Hits e misses no cache são cada vez mais importantes quando aumentamos o N. Pelo gráfico vemos que quanto maior o N, mais longe de $O(n^2)$ ficam os pontos (verdes e azuis).

No casos em que a taxa de hits é ótima para C e Fortran (caso i,j e j,i respectivamente), a diferença em tempo de execução difere de forma não tão impactante.

Um fator que complica a execução do programa para N grande é a quantidade de memória que precisa ser alocada para as 3 matrizes, que cresce rapidamente a ponto de consumir toda a memória física de um computador pessoal para N na ordem de 10^5.

Consumo de memória e CPU durante execução do *matrix_f*:



Conclusão: tomar cuidado com como uma determinada linguagem se relaciona com a estrutura de hardware e, quando programar pensando em alto desempenho, ter sempre em mente que diferenças visualmente simples no código, que mantém a semântica do programa, podem esconder grandes diferenças de desempenho em tempo de execução.

Código em C
matrix.c

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

void malloc_matrix(int ***array, int nrows, int ncolumns) {
  *array = malloc(nrows * sizeof(int *));
  if(*array == NULL) {
    fprintf(stderr, "out of memory\n");
    exit(1);
  }
  int i;
  for(i = 0; i < nrows; i++) {
    (*array)[i] = malloc(ncolumns * sizeof(int));
    if((*array)[i] == NULL) {
      fprintf(stderr, "out of memory\n");
      exit(1);
    }
  }
}

void zero_matrix(int ***array, int nrows, int ncolumns) {
  int i, j;
  for(i = 0; i < nrows; i++) {
    for(j = 0; j < ncolumns; j++) {
      (*array)[i][j] = 0;
    }
  }
}

int randint(int a, int b) {
  return a + (rand() % (b - a + 1));
}

void rand_matrix(int ***array, int nrows, int ncolumns) {
  int i, j;
  for(i = 0; i < nrows; i++) {
    for(j = 0; j < ncolumns; j++) {
      (*array)[i][j] = randint(-500, 500);
    }
  }
}

void init(int n, int ***v, int ***A, int ***x) {
  srand(time(NULL));
```

```
  malloc_matrix(v, n, 1);
  malloc_matrix(A, n, n);
  malloc_matrix(x, n, 1);

  zero_matrix(v, n, 1);
  rand_matrix(A, n, n);
  rand_matrix(x, n, 1);
}

void free_matrix(int ***array, int nrows) {
  int i;
  for(i = 0; i < nrows; i++) {
    free((*array)[i]);
  }
  free(*array);
  *array = NULL;
}

void finalize(int n, int ***v, int ***A, int ***x) {
  free_matrix(v, n);
  free_matrix(A, n);
  free_matrix(x, n);
}

double multiply_ij(int n, int **v, int **A, int **x) {
  clock_t start = clock();

  int i, j;
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      v[i][0] += A[i][j] * x[j][0];
    }
  }

  double elapsed_time = ((double)clock() - start) / CLOCKS_PER_SEC;
  return elapsed_time;
}

double multiply_ji(int n, int **v, int **A, int **x) {
  clock_t start = clock();

  int i, j;
  for (j = 0; j < n; j++) {
    for (i = 0; i < n; i++) {
      v[i][0] += A[i][j] * x[j][0];
    }
  }
```

```c
  double elapsed_time = ((double)clock() - start) / CLOCKS_PER_SEC;
  return elapsed_time;
}

double compute(int n, double (*multiply)(int, int **, int **, int **)) {
  int **v;
  int **A;
  int **x;
  init(n, &v, &A, &x);
  double elapsed_time = multiply(n, v, A, x);
  finalize(n, &v, &A, &x);
  return elapsed_time;
}

int main(int argc, char *argv[]) {
  if (argc != 4) {
    fprintf(stderr, "Usage: %s MIN MAX COUNT\n", argv[0]);
    exit(1);
  }

  int n;
  int min = atoi(argv[1]), max = atoi(argv[2]);
  int step = (max - min) / (atoi(argv[3]) - 1);

  printf("[1] Compute v = Ax looping through rows then columns (i, j)\n", step);
  printf("[2] Compute v = Ax looping through columns then rows (j, i)\n");
  printf("%6s\t%10s\t%10s\n", "N", "[1]", "[2]");
  for (n = min; n <= max; n += step) {
    printf("%6d\t%10f\t%10f\n", n, compute(n, &multiply_ij), compute(n, &multiply_ji));
  }

  return 0;
}
```

Compilação:
$ gcc -O3 matrix.c -o matrix_c

Execução:
$ ./matrix_c 1000 16000 11

Código em Fortran
matrix.f

```
program matrix
    integer, dimension(:,:), allocatable :: v, A, x

c    variables to measure time
    real t1, t2

    integer n, n_min, n_max, count, step
    integer i, j

c    read parameters from standard input
    character(len=10) :: arg
    if (iargc() .ne. 3) then
      write(*,*) "Required arguments: MIN MAX COUNT"
      call exit(1)
    end if

    call getarg(1, arg)
    read (arg,*) n_min
    call getarg(2, arg)
    read (arg,*) n_max
    call getarg(3, arg)
    read (arg,*) count
    step = (n_max - n_min) / (count - 1)

    write(*,*)"[1] Compute v = Ax looping through rows then columns
   &(i, j)"
    write(*,*)"[2] Compute v = Ax looping through columns then rows
   &(j, i)"
    write(*,'(A6,A10,A10)') "N    ","[1]    ","[2]"

c    main iteration
    do n = n_min, n_max, step

c    allocate memory for arrays
    allocate(v(n,1))
    allocate(A(n,n))
    allocate(x(n,1))

c    initialize arrays
    do i = 1, n
      v(i,1) = 0
      do j = 1, n
        A(i,j) = int(rand()*1000)-500
      end do
      x(i,1) = int(rand()*1000)-500
    end do

c    compute elapsed time
    t1 = compute_ij(n, v, A, x)
    t2 = compute_ji(n, v, A, x)
    write(*,'(I6,A1,F9.6,A1,F9.6)') n,'    ',t1,'    ',t2
```

```
c     deallocate memory
      deallocate(v)
      deallocate(A)
      deallocate(x)
      end do
      end

      real function compute_ij(n, v, A, x)
      integer, dimension(n,n) :: A
      integer, dimension(n,1) :: v, x
      real start_time, stop_time

      call cpu_time(start_time)

      do i = 1, n
        do j = 1, n
          v(i,1) = v(i,1) + A(i,j) * x(j,1)
        end do
      end do

      call cpu_time(stop_time)
      compute_ij = stop_time-start_time
      end

      real function compute_ji(n, v, A, x)
      integer, dimension(n,n) :: A
      integer, dimension(n,1) :: v, x
      real start_time, stop_time

      call cpu_time(start_time)

      do j = 1, n
        do i = 1, n
          v(i,1) = v(i,1) + A(i,j) * x(j,1)
        end do
      end do

      call cpu_time(stop_time)
      compute_ji = stop_time-start_time
      end
```

Compilação:
$ gfortran -O3 -g -march=native matrix.f -o matrix_f

Execução:
$ ./matrix_f 1000 16000 11