Assignment I: Performance Modeling & Benchmarking

Due Wednesday by 5pm **Points** 6 **Submitting** a file upload **File Types** pdf

Go through the exercise steps and answer questions that are marked **Question:**

- The submission is group-based. Join an empty Assignment Group in the Canvas People section even if you are working on your own. You **must** join a group.
- Submit a PDF. *No code submission* is required for this assignment, *but you must be ready to show and explain them.*
- The assignments can be completed either on Dardel or on your local computer. If you decide to complete the exercise on Dardel, read how to compile and run jobs **here**. For example how to allocate and run jobs with `salloc.`
- All the codes required for this assignment are also available at **https://github.com/KTH-HPC/DD2356/tree/master/Assignment-I** **(https://github.com/KTH-HPC/DD2356/tree/master/Assignment-I)**

Exercise 1. Modeling Sparse Matrix-Vector Multiply

In the lecture, we have seen how to develop a simple performance model for sparse-matrix multiplication. The goal of this exercise is to estimate the performance of sparse matrix-vector multiply for different sparse matrix sizes and compare it to actual performance measurements.

The code we use for this exercise is available **here** **(https://canvas.kth.se/courses/31107/files/5383787/download?download_frd=1)** . Compile it using the following:

```
$ cc -O2 -o spmv.out spmv.c
```

This code uses a sparse penta-diagonal (=5 non-zero diagonals and rest all zeros) matrix. A penta-diagonal matrix arises from the 2D finite-difference discretization of the Poisson equation and it is quite common to use these matrices in scientific computing.

Remember that the simple performance model of sparse matrix-vector multiply is the following:

$$Time = nnz(2c + 2.5r) + nrows(0.5r + w)$$

In this exercise, we will set the $nrows$ value and calculate $nnz = 5 * nrows$ since a penta-diagonal matrix has 5 non-zero values per row.

We will consider matrices with $nrows$ = 10^2, 10^4, 10^6, and 10^8.

Questions:

1. What is the performance in total execution time - do not consider data movement - according to your performance model on Dardel or your local computer for different sparse matrices $nrows$ = 10^2, 10^4, 10^6, and 10^8?
   - **Hint:** Use $Time = nnz * 2c$ and calculate c from the given clock speed of the processor in use (assuming also that one instruction is executed per cycle).
2. What is the **measured** performance in **total execution time** and **floating-point operations per second** running `spmv.c` for different sizes $nrows$ = 10^2, 10^4, 10^6, and 10^8? Compare the results from the performance model and experimental results. Discuss the comparison in the report.
   - **Note**: in `spmv.c`, we set up $nrows$ by setting $n = \sqrt{nrows}$.
   - **Hint:** use $nnz * 2$ and the total execution time to calculate the floating-point operations per second in SpMV.
3. What is the main reason for the observed difference between the modeled value and the measured value?
4. What are the read bandwidth values you measure running `spmv.c` for different sizes $nrows$ = 10^2, 10^4, 10^6, and 10^8?
   - **Hint:** use $nnz(sizeof(double) + sizeof(int)) + nrows(sizeof(double) + sizeof(int))$ from page 11 of **Lecture: Modeling Sparse Matrix Vector Multiply** and the total execution time to calculate the bandwidth of SpMV
5. What is the bandwidth you obtain by running the **STREAM benchmark** **(https://canvas.kth.se/courses/31107/files/5383788/download?download_frd=1)** on your system? How does it compare to the bandwidth you measured in SpMV? Discuss the comparison. Compile the STREAM benchmark with:

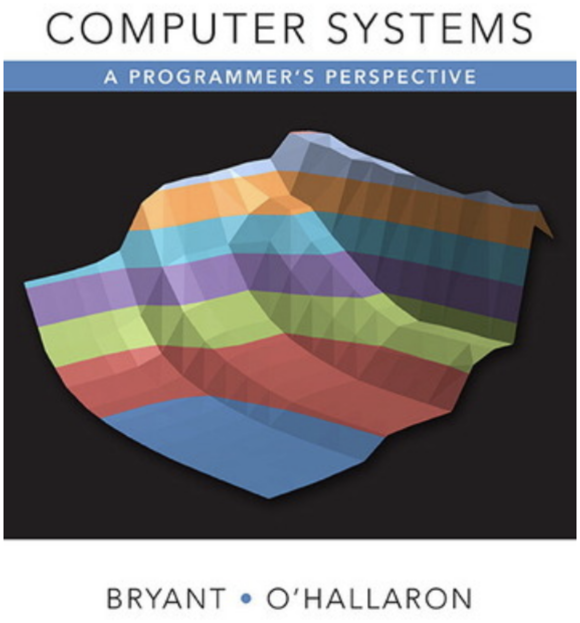
```
$ cc -O2 -o stream.out stream.c
```

Exercise 2. The Memory Mountain

This exercise can be done on a Dardel computing node or a local computer.

In this exercise, we measure the memory bandwidth of various levels of the memory hierarchy, and the impact of the (spatial and temporal) locality of accesses. You can use the code to prepare the "memory mountain" for a Dardel computing node processor.

By plotting the memory bandwidth depending on the size and stride, you will create the so-called memory mountain that is the cover of the book ***Computer Systems: A Programmer's Perspective* (https://csapp.cs.cmu.edu/)** by Randal E. Bryant and David R. O'Hallaron.



This program allocates a flat data buffer, walks through it, then computes the actual read bandwidth. The program runs the following loop and measures its execution time.

<u>Note: This program can be quite slow and will take 5+ minutes to run on a modern laptop.</u>

To install the code and run it, you will need to:

1. Download the code by loading the git repository

```
$ git clone https://github.com/KTH-HPC/DD2356.git
$ cd DD2356/Assignment-I/memory-mountain-example
```

Note: If you run this on Dardel you need to change the value in `#DEFINE MAXBYTES` to `(1 << 29)` ***in*** `mountain.c` ***before compiling. This might increase runtime.***

2. Build the code. Ignore the module swap command if you are running on your local computer.

```
$ module swap PrgEnv-cray PrgEnv-gnu
$ cd memory-mountain-example
$ make
```

3. You can try to run the benchmark, for example on Dardel:

```
$ salloc ...
$ srun -n 1 ./mountain.out
```

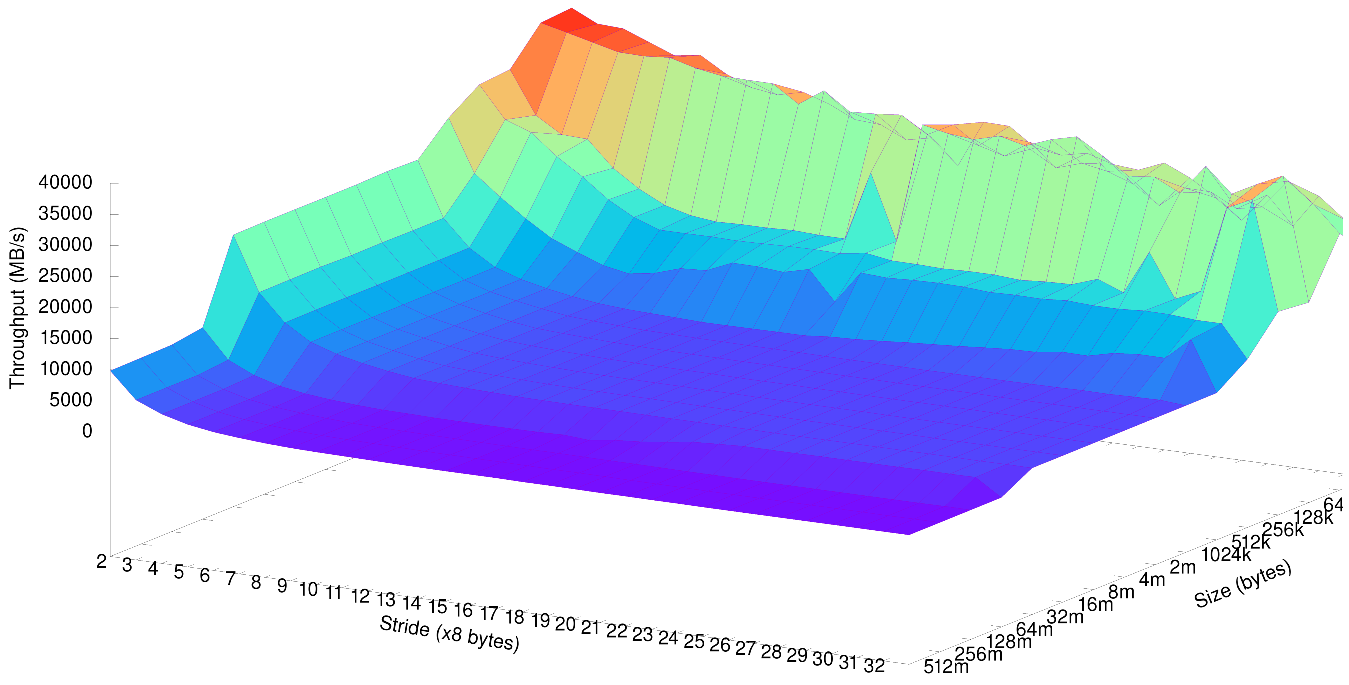
4. The program outputs a matrix that represents the bandwidth with respect to stride and data size.

Store the result with the following command, ignore salloc and srun if you are running on your laptop.

```
$ srun -n 1 ./mountain.out > results.txt
```

5. Transfer results back to your local computer, if you are running on Dardel, and visualize it using `plot.gplt` with Gnuplot. The script reads `results.txt` and generates a 3D figure. The resulting plot will be in `memory_mountain.png`. You can also run the commands in `plot.gplt` in the interactive mode of Gnuplot so you can rotate the 3D plot interactively. (if the script does not work with your system, most likely it is because of the shell scripting used to process the data file (e.g. `tail`), try to play around with it to make it work.)

```
$ gnuplot plot.gplt
```



Questions:

1. Report the name of the processor and the size of the L1, L2, and L3 of the processor you are benchmarking. You can check the specs of your processor online.
2. Create the memory mountain following the steps above.
   - Save the image of the displayed "memory mountain", and place the resulting image in your pdf file.
3. What region (array size, stride) gets the most **consistently** high performance (ignoring spikes in the graph that are noisy results...)? What is the read bandwidth reported?
4. What region (array size, stride) gets the most **consistently** low performance (Ignoring spikes in the graph that are noisy results...)? What is the read bandwidth reported?
5. When you look at the graph for stride=1, you (should) see relatively high performance compared

to stride=32. This is true even for large array sizes that are much larger than the L3 cache size. How is this possible, when the array cannot possibly all fit into the cache? Your explanation should include a brief overview of hardware prefetching as it applies to caches.
6. What is temporal locality? What is spatial locality?
7. Adjusting the total array size impacts temporal locality, why? Will an increased array size increase or decrease temporal locality?
8. Adjusting the read *stride* impacts spatial locality, why? Will an increased read stride increase or decrease spatial locality?
9. Repeat the analysis on your own laptop to check the differences. (Optional, if you did your analysis on Dardel)

Exercise 3. Write a Benchmark to Measure Performance

Modify the following provided **code** **(https://canvas.kth.se/courses/31107/files/5383715/download?download_frd=1)** and use it for completing the assignment. If you use Dardel, switch to the GNU environment (PrgEnv-gnu) instead of the default one (PrgEnv-cray) on Dardel. If you use your local computer, use the GNU compiler environment.

- **Question**: How do we switch the compiler environment on Dardel?

Using the gnu compiler environment will allow us to use different optimization flags.

The code is the following:

```
#define N 5000
#include <stdio.h>
#include <sys/time.h>

double mysecond();

int main(){
  int i, j;
  double t1, t2; // timers
  double a[N], b[N], c[N]; // arrays

  // init arrays
  for (i = 0; i < N; i++){
    a[i] = 47.0;
    b[i] = 3.1415;
  }

  // measure performance
  t1 = mysecond();
  for(i = 0; i < N; i++)
    c[i] = a[i]*b[i];
  t2 = mysecond();

  printf("Execution time: %11.8f s\n", (t2 - t1));
  return 0;
}

// function with timer
double mysecond(){
  struct timeval tp;
  struct timezone tzp;
  int i;

  i = gettimeofday(&tp,&tzp);
  return ( (double) tp.tv_sec + (double) tp.tv_usec * 1.e-6 );
}
```

1. Compiler the program in GNU environment with optimization flag `-O2` and execute it.
   1. **Question**: what is the average running time?
   2. **Question**: Increase N and compile the code, what is the average running time now?
2. Get the assembly instructions with

   ```
   $ cc -S -fverbose-asm -O2 benchmark.c
   ```

   More information about getting assembly language through GCC in **this tutorial**.
   1. **Question**: why is the execution time like that in the previous question when the flag `-O2` is used? Answer this question using the information you find in the assembly code.
   2. **Question**: What is the average execution time without the `-O2` compilation flag?
3. Check the tick (clock granularity) on Dardel or your local computer.
   - Download and compile this **code** on Dardel or your local computer.
   - Run from the command line giving it "100" (without quotation marks) as input (number of times the time period between the two execution time measurements are performed).
   - **Question**: What is the clock granularity on Dardel or your local computer.?
   - Replace the timer of the code using the RDTSC instruction as in **this code (https://github.com/skuhl/sys-prog-examples/blob/master/simple-examples/rdtsc.c)**. Read **this tutorial** to learn more about RDTSC instruction.

   - **Question**: What is the clock granularity when using the RDTSC timer?

4. Modify the program that you used in question 1.1 and do the following such that the code runs properly with `-O2` optimization:

   1. Avoid cold start issues by running the loop once before timing

2. Avoid clock granularity by timing multiple iterations of the same loop, then dividing by the number of outer iterations.
3. Trick a smart compiler by computing something with the result.
   - **Question (optional)**: Check if the code is running as expected by checking the assembly code.
   - **Question**: What are the minimum and average run times? Run the tests multiple times to avoid interference.

Exercise 4. Measure the Cache Usage in Matrix-Matrix Multiply

In this exercise, we use the PERF tool to measure the performance of a code for matrix-matrix multiply. You can use Dardel or your local computer for this exercise. In the case of your local computer, PERF must be supported.

A brief overview of PERF is given **here**.

Before starting the exercise, please take a look at **matrix_multiply.c** **(https://canvas.kth.se/courses/31107/files/5383729/download?download_frd=1)** . The program consists of three matrices A (`matrix_a`), B (`matrix_b`), and C (`matrix_c`). The three matrices are initialized and a warm-up run is executed. During the experiment, the program sets C to zero and calls `multiply_matrices()` to perform matrix multiplication. The test is repeated 10 times. After the tests, a checksum-like value is computed and the average running time per matrix multiplication is reported.

Switch to the GNU environment and compile the program with the following (use the exact environment and flags):

```
$ cc -g -O2 matrix_multiply.c -o matrix_multiply.out
```

The main focus of the program is:

```
void multiply_matrices()
{
  int i, j, k ;

  // Textbook algorithm
  for (i = 0 ; i < MSIZE ; i++) {

    for (j = 0 ; j < MSIZE ; j++) {
      for (k = 0 ; k < MSIZE ; k++) {
          matrix_r[i][j] += matrix_a[i][k] * matrix_b[k][j];
      }
    }
    }
  }
```

The innermost subscript `k` changes the fastest and the program steps sequentially through the memory elements of `matrix_a`. However, the program steps across the columns of `matrix_b`, resulting in a large physical stride through memory. (Recall that C arrays are arranged in row-major order in primary memory.) Access to `matrix_b` makes inefficient use of the L1 data cache.

The optimized code for the loop nest interchange program is

```
void multiply_matrices()
{
  int i, j, k ;
  // Loop nest optimized algorithm
  for (i = 0 ; i < MSIZE ; i++) {
    for (k = 0 ; k < MSIZE ; k++) {
      for (j = 0 ; j < MSIZE ; j++) {
        matrix_r[i][j] += matrix_a[i][k] * matrix_b[k][j];
      }
    }
  }
}
```

The nesting of the innermost loops has been changed. The index variables `j` and `k` change the most frequently and the access pattern through the two operand matrices is sequential using a small physical stride (8 bytes.) These changes improve access to memory data via the data cache.

Questions:

- Using *perf stat -e ...* and create a table reporting these quantities:
  - Attention! On Dardel, where do you put srun in this case, e.g. srun -n 1 perf or perf srun -n 1?

EVENT NAME	MSIZE = 64 Naive	MSIZE = 64 Optimized	MSIZE = 1000 Naive	MSIZE = 1000 Optimized
Elapsed time (seconds)	________	________	________	________
Instructions per cycle	________	________	________	________
L1 cache miss ratio	________	________	________	________
L1 cache miss rate PTI	________	________	________	________
LLC cache miss ratio	________	________	________	________
LLC cache miss rate				

PTI	____________	____________	____________	____________

- What are the factors that impact the most the performance of the matrix multiply operation for different matrix sizes and implementations (naive vs optimized)?

Exercise 5. Measure the Cache Usage in Transpose

In this exercise, we use the PERF tool to measure the performance of a **code for matrix transpose** **(https://canvas.kth.se/courses/31107/files/5383727/download?download_frd=1)** operation for different matrix sizes. The code can be compiled with:

```
$ cc -O2 -o transpose.out transpose.c
```

Questions:

- Using *perf stat -e ...* and create a table reporting these quantities:

EVENT NAME	N=64	N=128	N=2048	N=2049
Elapsed time (seconds)	____________	____________	____________	____________
Bandwidth/Rate (from the code and not PERF)	____________	____________	____________	____________
Instructions per cycle	____________	____________	____________	____________
L1 cache miss ratio	____________	____________	____________	____________
L1 cache miss rate PTI	____________	____________	____________	____________
LLC cache miss ratio	____________	____________	____________	____________
LLC cache miss rate PTI	____________	____________	____________	____________

- What are the factors that impact the performance of the transpose operation most for different matrix sizes and implementations?

- Which code transformations can be used in the code for the matrix transpose to improve the re-usage of cache?

Exercise 6. Vectorization

Questions:

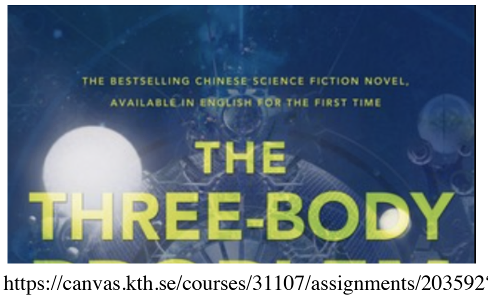
1. Find out how to request your compiler, e.g. GCC, Cray compiler … apply vectorization by checking on-line resources. For some systems and compilers, vectorization is the default with certain optimization flags.
    - Find out which optimization flag for your compiler includes vectorization.
2. Find out how you can get a report from the compiler about its success at vectorization.
    - In particular, find out which compiler flag enables a vectorization report for your compiler.
3. Read your compiler's documentation to find out what special directives or command-line options can affect vectorization
4. Obtain the vectorization report for the **matrix-matrix multiply code** **(https://canvas.kth.se/courses/31107/files/5383719/download?download_frd=1)** for MSIZE = 1000.
    - Which lines of the code are not vectorized if any, and in case why the compiler is not vectorizing them?

Bonus exercise: Performance Modeling of N-body Simulation

In an N-body problem, we need to find the positions and velocities of a collection of interacting particles over a period of time. For example, an astrophysicist might want to know the positions and velocities of a collection of stars, while a chemist might want to know the positions and velocities of a collection of molecules or atoms.

An N-body solver is a program that finds the solution to an n-body problem by simulating the behavior of the particles. The input to the problem is the mass, position, and velocity of each particle at the start of the simulation, and the output is the position and velocity of each particle at a sequence of user-specified times.

The N-body solver can be used to solve the three-body problem, like in the book we briefly discuss in the introductory lecture!


THE BESTSELLING CHINESE SCIENCE FICTION NOVEL,
AVAILABLE IN ENGLISH FOR THE FIRST TIME
THE
THREE-BODY

PROBLEM
CIXIN LIU
Translated by KEN LIU
"The best kind of science fiction, familiar but strange all at the same time."
—KIM STANLEY ROBINSON

Problem

We will write and parallelize a **two-dimensional** N-body solver that simulates the motions of planets or stars. We'll use Newton's second law of motion and his law of universal gravitation to determine the positions and velocities. Thus, if a particle q has a position $s_q(t)$ at time t, and particle k has a position $s_k(t)$, then the force on particle q exerted by particle k is given by

$$\mathbf{f}_{qk}(t) = -\frac{Gm_q m_k}{\left|\mathbf{s}_q(t) - \mathbf{s}_k(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_k(t)\right]$$

Here, G is the gravitational constant (6.673 × 10−11m3 /(kg ·s 2)), m_q and m_k are the masses of particles q and k, respectively. Also, the notation $s_q(t) - s_k(t)$ represents the distance from particle k to particle q. Note that in general the positions, the velocities, the accelerations, and the forces are vectors.

We can use the equation above to find the total force on any particle by adding the forces due to all the particles. If our n particles are numbered $0, 1, 2, \ldots, n-1$, then the total force on the particle q is given by

$$\mathbf{F}_q(t) = \sum_{\substack{k=0\\k\neq q}}^{n-1}\mathbf{f}_{qk} = -Gm_q\sum_{\substack{k=0\\k\neq q}}^{n-1}\frac{m_k}{\left|\mathbf{s}_q(t) - \mathbf{s}_k(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_k(t)\right]$$

The acceleration of an object is given by the second derivative of its position and that Newton's second law of motion states that the force on an object is given by its mass multiplied by its acceleration, so if the acceleration of particle q is $aq(t)$, then $Fq(t) = m_q a_q(t) = m_q s''q(t)$, where $s''q(t)$ is the second derivative of the position $s_q(t)$. Thus, we can use the equation above to find the acceleration of particle q :

$$\mathbf{s}_q''(t) = -G\sum_{\substack{j=0\\j\neq q}}^{n-1}\frac{m_j}{\left|\mathbf{s}_q(t) - \mathbf{s}_j(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_j(t)\right]$$

Therefore, Newton's laws give us a system of differential equations—equations involving derivatives

Therefore, Newton's laws give us a system of differential equations—equations involving derivatives —and our job is to find at each time t of interest the position $s_q(t)$ and velocity $v_q(t) = s'_q(t)$. We'll suppose that we either want to find the positions and velocities at the times

$$t = 0, \Delta t, 2\Delta t, \ldots, T\Delta t,$$

or, more often, simply the positions and velocities at the final time $T\Delta t$. Here, Δt and T are specified by the user, so the input to the program will be n , the number of particles, Δt, T, and, for each particle, its mass, its initial position, and its initial velocity.

Two Serial Programs

In outline, a serial n-body solver can be based on the following pseudocode:

```
1 Get input data;
2 for each timestep {
3     if (timestep output) Print positions and velocities of particles;
4     for each particle q
5        Compute total force on q;
6     for each particle q
7        Compute position and velocity of q;
8 }
9 Print positions and velocities of particles;
```

Simple Algorithm

We can use our formula for the total force on a particle to refine our pseudocode for the computation of the forces in Lines 4-5:

```
for each particle q {
    for each particle k != q {
            x_diff = pos[q][X] - pos[k][X];
            y_diff = pos[q][Y] - pos[k][Y];
            dist = sqrt(x_diff*x_diff + y_diff*y_diff);
            dist_cubed = dist*dist*dist;
            forces[q][X] -= G*masses[q]*masses[k]/dist_cubed * x_diff;
            forces[q][Y] -= G*masses[q]*masses[k]/dist_cubed * y_diff;
    }
}
```

Here, we are assuming that the forces and the positions of the particles are stored as two-dimensional arrays, forces, and pos, respectively. We're also assuming we've defined constants

$X = 0$ and $Y = 1$. So the x-component of the force on particle q is forces[q][X] and the y-component is forces[q][Y]. Similarly, the components of the position are pos[q][X] and pos[q][Y].

Reduced Algorithm

We can use Newton's third law of motion, that is, for every action there is an equal and opposite reaction, to halve the total number of calculations required for the forces. If the force on particle q due to particle k is f_{qk}, then the force on k due to q is $-f_{qk}$. Using this simplification we can modify our code to compute forces, as follows:

```
for each particle q
        forces[q] = 0;
for each particle q {
       for each particle k > q {
            x_diff = pos[q][X] - pos[k][X];
            y_diff = pos[q][Y] - pos[k][Y];
            dist = sqrt(x_diff*x diff + y_diff*y diff);
            dist_cubed = dist*dist*dist;
            force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
            force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff
            forces[q][X] += force_qk[X];
            forces[q][Y] += force_qk[Y];
            forces[k][X] -= force qk[X];
            forces[k][Y] -= force qk[Y];
       }
}
```

To better understand this pseudocode, imagine the individual forces as a two-dimensional array:

$$\begin{bmatrix} 0 & \mathbf{f}_{01} & \mathbf{f}_{02} & \cdots & \mathbf{f}_{0,n-1} \\ -\mathbf{f}_{01} & 0 & \mathbf{f}_{12} & \cdots & \mathbf{f}_{1,n-1} \\ -\mathbf{f}_{02} & -\mathbf{f}_{12} & 0 & \cdots & \mathbf{f}_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\mathbf{f}_{0,n-1} & -\mathbf{f}_{1,n-1} & -\mathbf{f}_{2,n-1} & \cdots & 0 \end{bmatrix}$$

Our original solver simply adds all of the entries in row q to get forces[q]. In our modified solver, when q = 0, the body of the loop for each particle q will add the entries in row 0 into forces[0]. It will also add the kth entry in column 0 into forces[k] for $k = 1, 2, \ldots, n-1$. In general, the q^{th} iteration will add the entries to the right of the diagonal (that is, to the right of the 0) in row q into forces[q], and the entries below the diagonal in column q will be added into their respective forces, that is, the kth entry will be added into forces[k]. Note that in using this modified solver, it's necessary to initialize the forces array in a separate loop, since the q^{th} iteration of the loop that calculates the forces will, in general, add the values it computes into forces[k] for $k = q+1, q+2, \ldots, n-1$, not just forces[q].

Particle Mover

The position and velocity remain to be found. We will use the following pseudocode

```
pos[q][X] += delta_t*vel[q][X];
pos[q][Y] += delta_t*vel[q][Y];
```

```
pos[q][Y] += delta_t*vel[q][Y];
vel[q][X] += delta_t/masses[q]*forces[q][X];
vel[q][Y] += delta_t/masses[q]*forces[q][Y];
```

Here, we're using pos[q], vel[q], and forces[q] to store the position, the velocity, and the force, respectively, of particle q.

Data structures

We are going to use an array type to store our vectors.

```
#define DIM 2
typedef double vect_t[DIM];
```

If forces are stored in an array using contiguous memory, we can use a fast function such as `memset` to quickly assign zeroes to all of the elements at the beginning of each iteration:

```
#include <string.h> /* For memset */
...
vect_t *forces = malloc(n*sizeof(vect_t));
...
for (step = 1; step <= n_steps; step++) {
  ...
  /* Assign 0 to each element of the forces array */
  forces = memset(forces, 0, n*sizeof(vect_t);
  for (part = 0; part < n-1; part++)
    compute_force(part, forces, ...);
}
```

Initialization

We provide you a simple code on how to initialize the bodies.

```
for each particle q {
    pos[q][X] = (rand() / (double)(RAND_MAX)) * 2 - 1;
    pos[q][Y] = (rand() / (double)(RAND_MAX)) * 2 - 1;

    vel[q][X] = (rand() / (double)(RAND_MAX)) * 2 - 1;
    vel[q][Y] = (rand() / (double)(RAND_MAX)) * 2 - 1;

    mass[q] = fabs((rand() / (double)(RAND_MAX)) * 2 - 1);
}
```

1. **Question**: Implement the simple and reduced algorithm of the N-Body simulator and answer the

following questions:

- What is the performance of the serial simple and reduced algorithms in execution time for 500, 1000, 2000 particles using 100 cycles and $\Delta t = 0.05$ with no output? Plot the performance results (including error bars) and discuss the results.
- Which algorithm has better cache reuse and why is that? Discuss the results. **Hint**: you can use *Perf* for monitoring cache counters or any tool you might find useful.

2. **Question**: Develop a performance model for the simple and reduced N-body codes:
   - Develop a performance model that takes as input the clock of the processor you are using, the number of particles and cycles and provides the execution time of the code as output assuming memory infinitely fast (**no cost for data movement**). Make a plot with the prediction values and compare them with the experimental results. Discuss the results in the report.
   - Develop a performance model that takes as input the clock of the processor and bandwidth of your system, the number of particles and cycles, and provides the execution time of the code as output **considering the cost of data movement**. Make a plot with the prediction values and compare them with the experimental results. Discuss the results in the report.

Assignment I: Performance Modeling & Benchmarking

Criteria	Ratings		Pts
Exercise I	**1 pts** **Full Marks**	**0 pts** **No Marks**	1 pts
Exercise II	**1 pts** **Full Marks**	**0 pts** **No Marks**	1 pts
Exercise III	**1 pts** **Full Marks**	**0 pts** **No Marks**	1 pts
Exercise IV	**1 pts** **Full Marks**	**0 pts** **No Marks**	1 pts
Exercise V	**1 pts** **Full Marks**	**0 pts** **No Marks**	1 pts
Exercise VI	**1 pts** **Full Marks**	**0 pts** **No Marks**	1 pts
Bonus Exercise	**1 pts** **Full Marks**	**0 pts** **No Marks**	1 pts
			Total Points: 7