20^{th} Marathon of Parallel Programming
SBAC-PAD/SSCAD – 2025

Calebe P. Bianchini[1] **and Luciano Gonda**[2]

[1]Mackenzie Presbyterian University

[2]Federal University of Mato Grosso do Sul

Rules for Remote Contest

For all problems, read the input and output session carefully. For all problems, a sequential implementation is provided, and it is against the output of those implementations that the output of your programs will be compared to decide if your implementation is correct. You can modify the program in any way you see fit, except when the problem description states otherwise. You must upload a compressed file (*zip*) containing your source code, the *Makefile*, and an execution script. The script must have the name of the problem. You can submit as many solutions to a problem as you want. Only the last submission will be considered. The *Makefile* must have the rule all, which will be used to compile your source code. The execution script runs your solution the way you design it – it will be inspected to ensure it does not corrupt the target machine.

The execution time of your program will be measured running it with time program and taking the real CPU time given. Each program will be executed at least three times with the same input, and the mean time will be taken into account. The sequential program provided will be measured in the same way. You will earn points in each problem, corresponding to the division of the sequential time by the time of your program (speedup). The team with the most points at the end of the marathon will be declared the winner.

This problem set contains 4 problems; pages are numbered from 01 to 08.

General Information

Compilation

You should use **CC** or **CXX** inside your *Makefile*. Be careful when redefining them! There is a simple *Makefile* inside the problem package that you can modify. Example:

```
FLAGS=-O3
EXEC=sum
CXX=g++

all: $(EXEC)

$(EXEC):
	$(CXX) $(FLAGS) $(EXEC).cpp -c -o $(EXEC).o
	$(CXX) $(FLAGS) $(EXEC).o -o $(EXEC)
```

Each judging machine has its group of compilers. See them below and choose well when writing your *Makefile*. The compiler that is tagged as *default* is predefined in **CC** and **CXX** variables.

machine	compiler	command
host	GCC 9.4.0 (*default*)	C = gcc C++ = g++
MPI	Open MPI 4.1.1 (*default*)	C = mpicc C++ = mpic++
	GCC 9.4.0	C = gcc C++ = g++
gpu	NVidia CUDA 12.0.1 (*default*)	C = nvcc C++ = nvcc
	GCC 8.3.1	C = gcc C++ = g++

Submitting

General information

You must have an execution script that has the same name of the problem. This script runs your solution the way you design it. There is a simple script inside you problem package that should be modified. Example:

```
#!/bin/bash
# This script runs a generic Problem A
# Using 32 threads and OpenMP
export OMP_NUM_THREADS=32
OMP_NUM_THREADS=32 ./sum
```

Submitting MPI

If you are planning to submit an MPI solution, you should compile using *mpicc*/*mpic++*. The script must call *mpirun*/*mpiexec* with the correct number of processes (max: 160).

```
#!/bin/bash
# This script runs a generic Problem A
# Using MPI in the entire cluster (4 nodes)
mpirun -np 4 ./sum
```

Comparing times & results

In your personal machine, measure the execution time of your solution using *time* program. Add input/output redirection when collecting time. Use *diff* program to compare the original and your solution results. Example:

```
$ time -p ./A < original_input.txt > my_output.txt
real 4.94
user 0.08
sys 1.56

$ diff my_output.txt original_output.txt
```

Do not measure time and **do not** add input/output redirection when submitting your solution - the *auto-judge system* is prepared to collect your time and compare the results.

Problem A
Hungry Ants

Miguel Felipe Silva Vasconcelos

Ants have the exceptional ability to find the shortest routes between the colony and a source of food. When they need to search for food, initially the ants take random paths and deposit pheromones along the way. Once a source of food has been found, the ant returns to the colony - and the path between the food source and the colony has pheromones. Given that the deposited pheromones evaporate over time, the shorter paths tend to accumulate stronger pheromone concentrations, since it is faster to travel through them, and the pheromones also attract more ants. As time passes, the ants will use a single path, as the pheromones of the other longer paths have evaporated. As a result, the colony collectively converges to the shortest route between the nest and the food source.

This natural behavior inspired the ACO algorithm, which is often applied to the Traveling Salesman Problem (TSP), in which a salesman must visit all cities exactly once and return to the starting point - used in real life for route planning services, for example.

At a high level, the ACO algorithm works as follows:

- Each ant builds its route and then computes its costs;
- The pheromones of all paths evaporates;
- The pheromone is deposited in the routes where the ants have passed.

Your task is to parallelize the sequential version of the ACO for the TSP problem. In what follows, there is a description of the input and expected output.

Input

The input is composed of three integer numbers separated by a space: CAI In which:

- C — Number of cities ($2 \leq C \leq 1000$);
- A — Number of ants ($1 \leq A \leq 200$);
- I — Number of iterations ($1 \leq I \leq 100$)

The input must be read from the standard input.

Output

For each input entry, the program must output two lines in the following format:

```
Best cost:  <float_value>
Signature:  <float_value>
```

The signature is a simple checksum computation with the order of the best path found, to reduce the output size and avoid printing all the cities of the best route.

The output must be written to the standard output.

Example

Sample Input 1	Sample output 1
`200 100 100`	`Best cost:  8098.4964` `Signature:  2.3000e+06`

Problem B
N Queens

Claudio Schepke

"The eight queens puzzle is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other; thus, a solution requires that no two queens share the same row, column, or diagonal. There are 92 solutions. The problem was first posed in the mid-19th century. In the modern era, it is often used as an example problem for various computer programming techniques"[1] due to its computational demands.

We provide an implementation that solves the problem for each size of the chessboard.

You are required to develop a parallel version based on programming models.

Input

The output contains a single line. It has the size N of the chessboard ($N \geq 4$).

The input must be read from the standard input.

Output

The output consists of all valid combinations of queen positions on the chessboard. Each line represents one combination formatted in a specific pattern – see the example below. All combinations are listed in alphabetical order.

The output must be written to the standard output.

Example

Sample input 1	Sample output 1
`4`	`SOLUTION: (1,2) (2,4) (3,1) (4,3)` `SOLUTION: (1,3) (2,1) (3,4) (4,2)`

[1] From `https://en.wikipedia.org/wiki/Eight_queens_puzzle`

Problem C
Reverse Time Migration (RTM)

Matheus Serpa

The Reverse Time Migration (RTM) application simulates wave propagation over time. Wave propagation is defined by the acoustic equation (Equation 1); distinct geological layers have different velocities (Equation 2), where $p(x, y, z, t)$ is the pressure at each point in the domain over time, $V(x, y, z)$ is the propagation velocity, and $\rho(x, y, z)$ is the density.

$$\frac{1}{V^2}\frac{\partial^2 p}{\partial t^2} = \nabla^2 p \tag{1}$$

$$\frac{1}{V^2}\frac{\partial^2 p}{\partial t^2} = \nabla^2 p - \frac{\nabla \rho}{\rho} \cdot \nabla p \tag{2}$$

The modeling simulates data acquisition in a seismic survey. At regular intervals, equipment mounted on a vessel emits waves that reflect and refract at subsurface interfaces. Eventually, these waves return to the sea surface and are recorded by microphones (hydrophones) attached to streamer cables towed by the ship. The set of signals received by each hydrophone over time constitutes a seismic trace. For each emission, the seismic traces of all hydrophones on the cable are recorded. The ship moves and emits signals over time.

The objective of this problem is to present a parallel version of the forward wavefield simulation.

Input

The input consists of the dimension N, which represents the $N \times N \times N$ cube to be simulated, and the number of time steps T to be simulated.

The input must be read from the standard input.

Output

The output is the listing of the positions in the cube considered mathematically relevant.

The output must be written to the standard output.

Examples

Sample input 1	Sample output 1
72 2	0.47445 -4.33689 22.83691 -93.79367 367.88351 0.47445 -4.33689 22.83691 -93.79367 367.88351 0.47445 -4.33689 22.83691 -93.79367 367.88351

Problem D
Cyclic Reduction Method for Tridiagonal Systems

Daniel Alfaro and Silvana Rossetto

Systems of linear equations appear very frequently in solving engineering and scientific problems. In general, a system of n linear equations with n unknowns can be written as:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \quad \cdots \quad + a_{1n}x_n &= d_1 \\ a_{21}x_1 + a_{22}x_2 + \quad \cdots \quad + a_{2n}x_n &= d_2 \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ a_{n1}x_1 + a_{n2}x_2 + \quad \cdots \quad + a_{nn}x_n &= d_n \end{cases} \tag{3}$$

In particular, tridiagonal systems of equations appear in many applications, such as in the calculation of *spline* functions and in solving boundary value problems for differential equations. When the coefficients of system (3) satisfy the conditions:

$$a_{ij} = 0 \quad \text{if} \quad |i - j| > 1,$$

we say that the system is tridiagonal. Therefore, in the coefficient matrix, all elements that are outside the main diagonal or the two diagonals immediately above and below the main diagonal are zero.

In this case, (3) can be rewritten as:

$$\begin{cases} b_1x_1 + c_1x_2 & & & = d_1, \\ \quad \ddots & & & \\ \quad a_ix_{i-1} + b_ix_i + c_ix_{i+1} & & & = d_i, \\ & \ddots & & \\ & & a_nx_{n-1} + b_nx_n & = d_n. \end{cases} \tag{4}$$

The **cyclic reduction method** is used to solve systems of tridiagonal equations[2]. This method can also be applied when the system matrix is block-tridiagonal and stands out for its simplicity and ease of parallelization.

The main idea of the cyclic reduction method consists of eliminating half of the variables from the system, regrouping the equations into a system with only half of the unknowns, and continuing to repeat this process until arriving at a system with only two unknowns. This procedure can be developed in a simple form for the case of a tridiagonal system, since the unknowns can be eliminated in such a way that the new system (with half the variables) is also tridiagonal. Thus, to obtain the solution of the initial system with n unknowns, it is sufficient to solve a reduced system containing only $n/2$ unknowns and then apply a simple substitution process to calculate the remaining $n/2$ unknowns.

[2]A full description of that problem can be found here: `https://www.sbmac.org.br/wp-content/uploads/2022/08/livro_84.pdf`, section 3.2

Applying these processes recursively, we arrive at the *cyclic reduction method*. To simplify our notation, we will consider the case where $n = 2^p$ with $p \geq 1$. The method consists of two stages: reduction and substitution.

During the reduction, in the first iteration, we obtain a reduced tridiagonal system containing the $n/2$ equations E'_{2j} for the unknowns x_{2j} where $j = 1, 2, \ldots, n/2$. In the second iteration, from the reduced system resulting from the first iteration, we obtain a new reduced system with $n/4$ equations E''_{4j} for the unknowns x_{4j} where $j = 1, 2, \ldots, n/4$. Finally, after $p-1$ iterations, as a result of the reduction process, we obtain a system with two equations for the unknowns $x_{n/2}$ and x_n.

The substitution stage begins after calculating $x_{n/2}$ and x_n. In the first iteration of this stage, we calculate the two unknowns $x_{n/4}$ and $x_{3n/4}$ by substituting $x_{n/2}$ and x_n into the reduced system obtained in iteration $p-2$. In the next iteration, we substitute the four previously obtained unknowns $x_{n/4}$, $x_{n/2}$, $x_{3n/4}$, and x_n into the reduced system obtained in the $(p-3)$-th iteration of the reduction stage to calculate four more unknowns: $x_{n/8}$, $x_{3n/8}$, $x_{5n/8}$, and $x_{7n/8}$. By continuing this process, after completing the $(p-1)$-th iteration of the substitution stage, we will compute all the unknowns of the initial system.

We can notice that the transformations of the equations to be performed in each iteration of the reduction stage can be done in parallel, but all calculations corresponding to one iteration must be completed before starting the next iteration. A similar observation can be made regarding the substitution stage.

For simplicity, we will consider the case where the dimension of the system is a power of 2 with exponent $p \geq 2$.

Input

The program must receive as input the exponent $p \geq 2$, such that the dimension of the system will be given by 2^p; the vectors of the coefficients of the diagonals (`a`, `b`, `c`); and the vector of the constant term (`d`).

The input must be read from the standard input.

Output

The output must be the vector of unknowns (`x`).

The output must be written to the standard output.

Examples

Sample input 1	Sample output 1
2 0 -1 -1 -1 3 3 3 3 -1 -1 -1 0 2 1 1 2	1.000000 1.000000 1.000000 1.000000

Sample input 2
3 0.000000 -1.125000 -1.250000 -1.375000 -1.500000 -1.625000 -1.750000 -1.875000 3.000000 3.125000 3.250000 3.375000 3.500000 3.625000 3.750000 3.875000 -1.000000 -1.000000 -1.000000 -1.000000 -1.000000 -1.000000 -1.000000 0.000000 1.000000 2.125000 3.250000 4.375000 5.500000 6.625000 7.750000 17.875000
Sample output 2
1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000