Maratona de Programação Paralela
ERAD/RS – 2025

Matheus S. Serpa[1] **e Calebe P. Bianchini**[2]

[1]Universidade Federal do Rio Grande do Sul

[2]Universidade Presbiteriana Mackenzie

Regras Gerais

Para todos os problemas será dada uma implementação sequencial. É contra a saída dessas implementações que a saída dos seus programas será comparada para decidir se a sua implementação está correta. Você pode modificar o programa da maneira que achar adequada, exceto quando a descrição do problema indicar o contrário. Você deve enviar um arquivo compactado (*zip*) com seu código fonte, o *Makefile* e um script de execução. O script deve ter o nome do problema. Você pode enviar quantas soluções para um problema desejar. Somente a última submissão será considerada. O *Makefile* deve possuir a regra `all`, que será utilizada para compilar seu código fonte. O script de execução executa sua solução da maneira que você a projetou — ela será inspecionada para não corromper a máquina de destino.

O tempo de execução do seu programa será medido executando-o com o programa *time* e considerando o tempo real de CPU fornecido. Cada programa será executado pelo menos três vezes com a mesma entrada e o tempo médio será levado em consideração. O programa sequencial fornecido será medido da mesma maneira. Você ganhará pontos em cada problema, correspondentes à divisão do tempo sequencial pelo tempo do seu programa (*speedup*). A equipe que somar mais pontos ao final da maratona será declarada vencedora.

Este caderno de problemas contém 3 problemas; as páginas estão numeradas de 01 **à** 05**.**

Informações Gerais

Compilação

Você deve usar **CC** ou **CXX** dentro do seu *Makefile*. Tenha cuidado ao redefini-los! Há um *Makefile* simples dentro do pacote do seu problema que você pode modificar. Exemplo:

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

all: $(EXEC)

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

Cada máquina de julgamento possui seu grupo de compiladores. Veja-os abaixo e escolha bem ao escrever seu *Makefile*. O compilador marcado como *default* é o predefinido nas variáveis **CC** e **CXX**.

machine	compiler	command
host	GCC 13.2.0 (*default*)	C = gcc C++ = g++
MPI	OpenMPI 5.0.0 (*default*)	C = mpicc C++ = mpic++
gpu	NVidia CUDA release 11.6, V11.6 (*default*)	C = nvcc C++ = nvcc

Submissão

Informações gerais

Você deve ter um script de execução com o mesmo nome do problema. Esse script executa sua solução da maneira como você a desenvolveu. Existe um exemplo de script simples dentro do pacote do seu problema que deve ser modificado. Exemplo:

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

Submissão com MPI

Se você pretende submeter uma solução usando MPI, deve compilar utilizando *mpicc*/*mpic++*. O script deve chamar *mpirun*/*mpiexec* com o número correto de processos (máximo: 160).

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

Comparando tempos & resultados

Na sua máquina pessoal, meça o tempo de execução da sua solução utilizando o programa *time*. Adicione redirecionamento de entrada/saída ao coletar o tempo. Use o programa *diff* para comparar os resultados originais com os resultados da sua solução. Exemplo:

```
$ 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
```

Não meça o tempo e **não** adicione redirecionamento de entrada/saída ao submeter sua solução – o *sistema auto-judge* está preparado para coletar seu tempo e comparar os resultados.

Problema A
k-Nearest Neighbors Classifier

Dado um conjunto de pontos bidimensionais S dividido em n grupos, um inteiro k e um ponto bidimensional P a ser classificado, o algoritmo dos *k-Nearest Neighbors Classifier* (KNN) pode ser utilizado para resolver o problema de classificação do ponto P, ou seja, classificar o ponto P em um dos n grupos existentes com base em algum critério.

O algoritmo KNN funciona comparando as distâncias de cada ponto $s \in S$ até o ponto P, então seleciona os k pontos com os menores valores de distância e conta a frequência de cada grupo nesses k pontos. O ponto P é então classificado como pertencente ao grupo com a maior frequência entre os k vizinhos mais próximos.

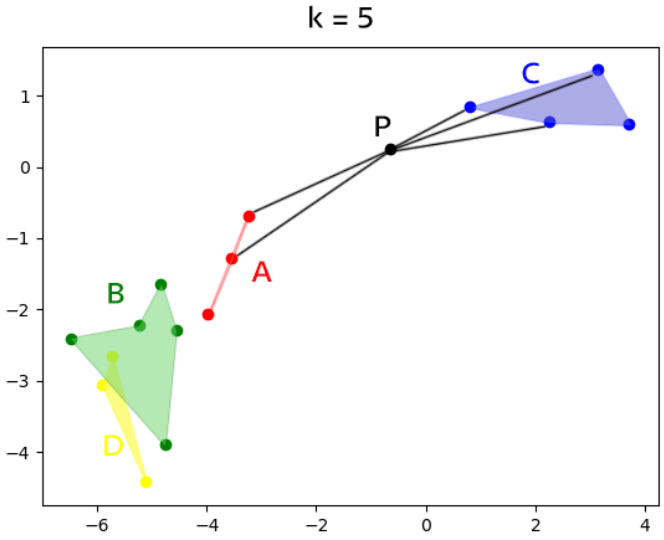

Figura 1. Exemplo simples da classificação com KNN.

A Figura 1 ilustra um exemplo da aplicação do algoritmo de classificação KNN. Os pontos foram separados em 4 grupos diferentes (*isto é*, A, B, C e D) e um ponto P sem grupo é fornecido para ser classificado. O algoritmo KNN então busca os k pontos mais próximos do ponto P e, em seguida, os grupos têm suas frequências contadas, sendo a maior frequência escolhida como a classificação de P. No caso deste exemplo, o ponto P é classificado como pertencente ao grupo C.

Entrada

A entrada representa a classificação de um único ponto. A primeira linha contém o padrão "n_groups=n", onde n é o número total de grupos nesta entrada. Em seguida, para cada um dos n grupos, serão fornecidas as seguintes linhas: a primeira linha tem o padrão "label=c", onde c é um único caractere que representa o rótulo (*isto é*, a identificação) do grupo; a segunda linha segue o padrão "length=L", onde L é o número de pontos bidimensionais neste grupo; e então, as próximas L linhas contêm o padrão "(x,y)", que representa cada ponto pertencente ao grupo mencionado. Após todos os grupos terem sido

representados, há uma única linha com o padrão "k=k", onde k é o parâmetro discutido anteriormente. Por fim, a última linha também contém o padrão "(x,y)", representando as coordenadas do ponto bidimensional a ser classificado.

Os dados de entrada devem ser lidos da entrada padrão.

Saída

A saída contém um único caractere, sendo ele o rótulo do grupo escolhido como classificação para o ponto fornecido.

A saída deve ser escrita na saída padrão.

Exemplo

Entrada	**Saída**
n_groups=4 label=A length=3 (-3.55,-1.28) (-3.99,-2.06) (-3.23,-0.70) label=B length=5 (-4.85,-1.65) (-5.23,-2.22) (-4.75,-3.89) (-6.48,-2.41) (-4.56,-2.29) label=C length=4 (2.25,0.64) (0.80,0.85) (3.13,1.37) (3.71,0.59) label=D length=3 (-5.73,-2.65) (-5.11,-4.41) (-5.92,-3.06) k=5 (-0.65,0.25)	C

Problema B
Caminho com Maior Soma em uma Árvore Genérica

Dada uma árvore genérica, seu objetivo é identificar e calcular o caminho da raiz até uma folha que possua a maior soma dos valores dos nós. Cada nó da árvore possui um valor associado, e um caminho é definido como uma sequência de nós que vai da raiz até uma folha.

Sua tarefa é desenvolver um programa paralelo para resolver este problema.

Entrada

A primeira linha da entrada contém um único número inteiro N ($1 \leq N < 1500000$), o número de nós da árvore. As próximas N linhas descrevem os nós. Cada linha começa com um valor decimal V ($0.1 \leq V \leq 100.0$), que representa o valor do nó, seguido por um número inteiro K ($0 \leq K \leq 1000$), que indica a quantidade de filhos. Em seguida, há K inteiros representando os índices dos filhos (começando por 1).

A entrada deve ser lida da entrada padrão.

Saída

A primeira linha da saída deve conter o maior valor decimal encontrado. A segunda linha deve conter os índices que compõem o caminho da raiz até a folha com a maior soma. Os índices devem ser separados por espaços e iniciar pelo valor 1.

A saída deve ser escrita na saída padrão.

Exemplo

Entrada exemplo 1	Saída exemplo 1

`5`
`10.5 2 2 3`
`5.2 2 4 5`
`3.3 0`
`4.1 0`
`2.4 0`
	`Soma Máxima:  19.80`
	`Caminho:  1 2 4`

Problema C
Zeros da Função Zeta de Riemann

Os zeros da função zeta de Riemann, denotada por $\zeta(s)$, são os valores da variável complexa s para os quais $\zeta(s) = 0$. Existem dois tipos de zeros: triviais e não-triviais.

Os zeros triviais ocorrem nos inteiros pares negativos $s = -2, -4, -6, ...$ Eles são considerados "triviais" porque sua existência é relativamente fácil de demonstrar usando a equação funcional da função zeta.

Os zeros não-triviais são os valores complexos de s para os quais $\zeta(s) = 0$ e cuja parte real está entre 0 e 1. Eles são chamados de "não-triviais" porque sua distribuição é menos compreendida, e seu estudo é importante para entender os números primos e objetos relacionados na teoria dos números.

A Hipótese de Riemann é um dos problemas mais famosos e antigos ainda não resolvidos da matemática, proposta por Bernhard Riemann em 1859. Ela conjectura que a função zeta de Riemann possui zeros apenas nos inteiros pares negativos e em números complexos cuja parte real é $1/2$.

O código relacionado a este desafio calcula o número de zeros sobre a linha crítica da função Zeta. O objetivo não é calcular os zeros: nós os contamos para verificar se estão sobre a Linha de Riemann.

O exercício consiste em amostrar uma região da linha crítica para contar quantas vezes a função muda de sinal, de forma que haja pelo menos um zero entre dois pontos de amostragem. Aqui utilizamos uma amostragem constante, mas você pode reimplementar livremente a maneira de proceder.

Escreva uma versão paralela desta solução.

Entrada

Na primeira linha da entrada, você receberá três números inteiros: o primeiro, L, representa o limite inferior; o segundo, U, representa o limite superior; e o último, S, representa quantas amostras serão testadas.

A entrada deve ser lida da entrada padrão.

Saída

A saída contém uma única linha. Ela deve conter o número total de zeros encontrados.

A saída deve ser escrita na saída padrão.

Exemplos

Entrada exemplo 1	Saída exemplo 1
`10 1000 100`	`Foram encontrados 649 Zeros`