Maratona de Programação Paralela: Relatório das soluções

Novembro 2023 - Feito em LaTeX.

- **Aluno:**
  - Lucas Mateus de Moraes (RA: 22.220.004-0)
- **Professor:** Dr. Calebe de Paula Bianchini

1 Descrição do experimento:

A entrega de relatório da atividade consistia em 4 problemas de paralelismo que podiam ser resolvidos de forma distribuida, um problema deveria ser escolhido para ser apresentado com uma solução, no presente relatório o problema escolhido para ser solucionado foi o problema de letra **"B"** (*Highest Sum Path in Generic Tree*).

Os outros problemas serão abordados com possíveis soluções apenas descritas em texto. Os códigos dos problemas e da solução podem ser consultados no repositório disponível no *link* abaixo:

- https://github.com/luca-moraes/sistemasDistribuidosMaratonaProblemasParalelismo

2 Problema A (*Number of Submatrices that Sum to k*):

2.1 Descrição do Problema:

O Problema A consiste em contar o número de submatrizes não vazias em uma matriz M × N de inteiros, onde a soma dos elementos dessas submatrizes é igual a um valor dado k. A matriz pode conter números negativos e as submatriz possíveis obtidas ao remover alguma linhas ou coluna devem ser consideradas.

2.2 Solução Serial:

Uma solução serial para este problema envolve iterar sobre todas as submatrizes possíveis da matriz dada, calcular a soma de cada submatriz e verificar se a soma é igual a k. Esta abordagem é geralmente ineficiente para grandes conjuntos de dados, já que a complexidade de tempo é elevada.

2.3 Solução com Paralelismo:

Uma solução paralela pode ser alcançada dividindo a matriz em partes menores e distribuindo o trabalho de verificação da soma entre essas partes para processos diferentes ou várias *threads*. Isso pode ser feito de forma eficiente utilizando técnicas de programação paralela, como dividir e conquistar. Cada processador em sistemas distribuídos, usando o MPI por exemplo, ou no mesmo processador com *threads* diferentes, lida com uma parte da matriz e verifica as submatrizes, reduzindo assim o tempo total de execução.

2.4 Conclusão:

O problema de contar o número de submatrizes com soma igual a k em uma matriz pode ser abordado tanto de forma serial quanto paralela. A solução serial é simples, mas pode ser ineficiente para grandes conjuntos de dados. A solução paralela, por outro lado, aproveita a capacidade de processamento simultâneo de vários núcleos ou processos em máquinas com MPI, proporcionando potencialmente um desempenho melhor em termos de tempo de execução, especialmente para matrizes de grande porte.

3 Problema B (*Highest Sum Path in Generic Tree*):

3.1 Descrição do Problema:

O Problema B envolve encontrar e calcular o caminho, em uma árvore genérica, que vai da raiz a uma folha e tem a maior soma dos valores dos nós. Cada nó na árvore possui um valor associado, e um caminho é definido como uma sequência de nós da raiz a uma folha.

3.2 Solução Serial:

Uma solução serial para este problema seria implementar um algoritmo de busca em profundidade (DFS) para percorrer a árvore, calculando a soma dos caminhos da raiz a cada folha. Durante esse processo, o algoritmo manteria o controle do caminho com a maior soma. Ao final, a resposta seria a soma máxima e o caminho correspondente.

3.3 Solução com Paralelismo:

Uma solução paralela poderia envolver a divisão da árvore em subárvores e atribuir diferentes processadores ou *threads* para calcular a soma dos caminhos em paralelo. Cada subárvore seria tratada como uma tarefa independente, e a resposta final seria calculada considerando os resultados parciais de cada subárvore.

3.4 Código da solução fornecido no desafio:

Listing 1: Código fornecido no desafio

```c
// Lucas Mateus de Moraes - RA: 22.220.004-0

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

#define MAX_CHILDREN 2
#define MAX_VALUE 100
#define MAX_NODES 1000

typedef struct {
    double value;
```

```c
    int num_children;
    int children[MAX_CHILDREN];
} Node;

typedef struct {
    double sum;
    int path[MAX_NODES];
    int pathLength;
} Result;



Result computePaths(Node* tree, int idx) {
    Result res = {0, {0}, 0};

    if (idx == -1) return res;

    if (tree[idx].num_children == 0) {
        res.sum = tree[idx].value;
        res.path[res.pathLength++] = idx;
        return res;
    }

    double maxSum = -1;
    Result maxChildRes;

    for (int i = 0; i < tree[idx].num_children; ++i) {
        Result childRes = computePaths(tree, tree[idx].children[i]);
        if (childRes.sum > maxSum) {
            maxSum = childRes.sum;
            maxChildRes = childRes;
        }
    }

    res.sum = tree[idx].value + maxSum;
    res.path[0] = idx;
    for (int i = 0; i < maxChildRes.pathLength; ++i) {
        res.path[i + 1] = maxChildRes.path[i];
    }
    res.pathLength = maxChildRes.pathLength + 1;

    return res;
}


int main() {
    int N;
    scanf("%d", &N);

    Node* tree = (Node*)malloc(N * sizeof(Node));
    for (int i = 0; i < N; ++i) {
        scanf("%lf %d", &tree[i].value, &tree[i].num_children);
        for (int j = 0; j < tree[i].num_children; ++j) {
            scanf("%d", &tree[i].children[j]);
            tree[i].children[j]--;
        }
    }

    Result finalRes = computePaths(tree, 0);
    printf("Max Sum: %.2lf\n", finalRes.sum);
    printf("Path: ");
    for (int i = 0; i < finalRes.pathLength; ++i) {
        printf("%d ", finalRes.path[i] + 1);
    }
    printf("\n");
    free(tree);
    return 0;
}
```

3.5 Código da solução com implementação do MPI:

Listing 2: Código com implementação do MPI

```
// Lucas Mateus de Moraes - RA: 22.220.004-0

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

#define MAX_CHILDREN 2
#define MAX_VALUE 100
#define MAX_NODES 1000

typedef struct {
    double value;
    int num_children;
    int children[MAX_CHILDREN];
} Node;

typedef struct {
    double sum;
    int path[MAX_NODES];
    int pathLength;
} Result;

Result computePaths(Node* tree, int idx) {
    Result res = {0, {0}, 0};

    if (idx == -1) return res;

    if (tree[idx].num_children == 0) {
        res.sum = tree[idx].value;
        res.path[res.pathLength++] = idx;
        return res;
    }

    double maxSum = -1;
    Result maxChildRes;

    #pragma omp parallel for
    for (int i = 0; i < tree[idx].num_children; ++i) {
        Result childRes = computePaths(tree, tree[idx].children[i]);
        #pragma omp critical
        {
            if (childRes.sum > maxSum) {
                maxSum = childRes.sum;
                maxChildRes = childRes;
            }
        }
    }

    res.sum = tree[idx].value + maxSum;
    res.path[0] = idx;
    for (int i = 0; i < maxChildRes.pathLength; ++i) {
        res.path[i + 1] = maxChildRes.path[i];
    }
    res.pathLength = maxChildRes.pathLength + 1;

    return res;
}

int main(int argc, char** argv) {
    int N;
    scanf("%d", &N);

    Node* tree = (Node*)malloc(N * sizeof(Node));
    for (int i = 0; i < N; ++i) {
        scanf("%lf %d", &tree[i].value, &tree[i].num_children);
        for (int j = 0; j < tree[i].num_children; ++j) {
```

```
            scanf("%d", &tree[i].children[j]);
            tree[i].children[j]--;
        }
    }

    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    Result localRes = computePaths(tree, 0);
    Result globalRes;

    MPI_Reduce(&localRes, &globalRes, 1, MPI_DOUBLE_INT, MPI_MAXLOC, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Max Sum: %.2lf\n", globalRes.sum);
        printf("Path: ");
        for (int i = 0; i < globalRes.pathLength; ++i) {
            printf("%d ", globalRes.path[i] + 1);
        }
        printf("\n");
    }

    free(tree);
    MPI_Finalize();
    return 0;
}
```

3.6 Conclusão:

4 Problema C (*NPbonacci*):

4.1 Descrição do Problema:

O Problema C envolve a computação de termos em uma sequência chamada *NPbonacci*, que é uma generalização da sequência de Fibonacci. Na sequência *NPbonacci*, os termos são construídos a partir de uma lista de base com tamanho N e uma lista de pesos P. A computação de cada termo leva em consideração os últimos N termos na sequência, multiplicando-os pelos pesos correspondentes na lista P e somando os resultados ponderados. A tarefa é calcular o termo FbH modulo $10^9 + 7$, dados os valores de N, H, as condições iniciais A e a lista de pesos P.

4.2 Solução Serial:

Uma solução serial para este problema envolveria a implementação direta do algoritmo de *NPbonacci*, calculando os termos sequencialmente até atingir o termo desejado F bH. Isso seria feito iterativamente, levando em consideração as condições iniciais e aplicando a ponderação apropriada.

4.3 Solução com Paralelismo:

Para uma solução paralela, podemos explorar a natureza recursiva da sequência e distribuir o cálculo dos termos em paralelo. Cada processo ou *thread* poderia ser responsável por calcular uma parte da sequência, reduzindo assim o tempo total de execução. Técnicas de programação paralela, como divisão e conquista, podem ser aplicadas para otimizar o desempenho.

4.4 Conclusão:

O problema de calcular termos na sequência *NPbonacci* pode ser abordado de forma serial ou paralela. A solução serial é direta, mas pode ser ineficiente para valores grandes de H. A solução paralela, por outro lado, aproveita o poder de processamento simultâneo de vários núcleos ou processadores, oferecendo potencialmente um desempenho melhor, especialmente para casos em que H é grande.

5 Problema D (*Number of ways to traverse a grid*):

5.1 Descrição do Problema:

O Problema D envolve contar o número de maneiras possíveis de percorrer uma matriz quadrada NxN, começando da célula (0,0) e terminando na célula (N-1, N-1). Cada movimento só pode ser realizado indo para a direita (aumentando a coordenada x) ou para cima (aumentando a coordenada y). Além disso, o problema apresenta a restrição de que algumas células estão bloqueadas, e nenhum caminho pode atravessá-las. O algoritmo base proposto utiliza programação dinâmica para resolver o problema.

5.2 Solução Serial:

Uma solução serial seria implementar um algoritmo para calcular o número de caminhos possíveis. O algoritmo iteraria pela matriz, considerando os caminhos possíveis de célula em célula, evitando as células bloqueadas. A resposta seria então calculada considerando o número total de caminhos válidos, levando em conta todas as possibilidades, o que pode tornar custosos conforme o tamanho do problema cresce.

5.3 Solução com Paralelismo:

Para uma solução paralela, poderíamos dividir a matriz em submatrizes menores e atribuir diferentes processadores ou *threads* para calcular o número de caminhos em paralelo para essas submatrizes. Técnicas de divisão e conquista poderiam ser aplicadas para otimizar o processo paralelo. Isso poderia resultar em uma redução significativa no tempo de execução, especialmente para matrizes grandes.

5.4 Conclusão:

O problema de contar o número de caminhos em uma matriz, com algumas células bloqueadas, pode ser abordado tanto de forma serial quanto paralela. A solução serial pode ser ineficiente para matrizes grandes. A solução paralela, por outro lado, aproveita o paralelismo para calcular várias partes da matriz simultaneamente, proporcionando potencialmente um desempenho melhor, especialmente para matrizes com grandes quantidades de linhas e colunas.