Implementação do Algoritmo *Floyd* Utilizando Abordagem Híbrida de Computação Paralela

Carlos Alberto Franco Maron
Pontifícia Universidade Católica do Rio Grande do Sul
Porto Alegre – Brasil
carlos.maron@acad.pucrs.br

***Resumo*—A programação paralela é um paradigma que propõe resolver problemas computacionais em pouco tempo de processamento e utilizando de maneira eficiente os recursos disponíveis. Bibliotecas de programação paralela permitem a decomposição destes problemas através do uso de memória distribuída (MPI - *Message Passing Interface*) e memória compartilhada (OMP - *Open Multi-Processing*). Computação híbrida combina essas duas formas de paralelização com a proposta de acelerar mais este processo computacional. Desta forma, este trabalho busca utilizar as bibliotecas OMP e MPI combinadas para resolução de um problema clássico da computação de alto desempenho. No trabalho, foi utilizado o algoritmo de *Floyd* paralelizado em um *cluster* composto com 4 nodos.**

***Keywords*—Programação paralela, MPI, OpenMP, Algoritmo Floyd.**

I. Introdução

O algoritmo de *Floyd* foi publicado originalmente em 1962 por Robert Floyd, com a proposta de resolver o problema de cálculo do caminho mais curto entre todos os pares de um grafo orientado e valorado [1].

Este algoritmo tem complexidade de tempo $\theta(N^3)$, e envolve a programação dinâmica para resolução de problemas de otimização combinatória, aplicável principalmente em problemas nos quais a solução pode ser computada a partir de uma solução previamente calculada e memorizada. Esse tipo de otimização evita o recálculo de operações já realizadas e de outros subproblemas que, sobrepostos, compõem o problema original. Fazendo uma análise com o padrão de programação divisão e conquista, o qual divide o total do problema em subproblemas menores, resolvendo estes em instâncias menores, e ao final realiza a combinação de todos resultados processados. Portanto, a programação dinâmica procura evitar operações desnecessárias reutilizando os cálculos de subproblemas já realizados, os quais geralmente são previamente armazenados. Outros exemplos de algoritmos que fazem uso da computação dinâmica são *Fibonacci* e multiplicação de cadeia de matrizes [2].

O desenvolvimento de algoritmos que envolvem programação dinâmica segue uma sequência de 4 etapas: (**I**) caracterizar a estrutura de uma solução ótima. (**II**) definir recursivamente o valor de uma solução ótima. (**III**) calcular o valor de uma solução ótima, normalmente de baixo para cima (*bottom up*). (**IV**) construir uma solução ótima com as informações calculadas [2].

Neste trabalho, é apresentado a implementação paralela híbrida do algoritmo de *Floyd*. Na seção II é apresentando brevemente o funcionamento do algoritmo, bem como suas aplicações. Na Seção III são descritos os procedimentos realizados na implementação e nos testes. Na Seção IV são apresentados os resultados de desempenho do algoritmo paralelizado. A Seção V apresenta um breve relato das dificuldades da implementação e, por fim a conclusão do trabalho é apresentada na Seção VI.

II. Algoritmo de *Floyd*

O algoritmo de *Floyd* recebe como entrada uma matriz de adjacência que representa um grafo (V, E) orientado e valorado. O valor de um caminho entre dois vértices é a soma dos valores de todas as arestas ao longo desse caminho. As arestas E do grafo podem ter valores negativos, mas o grafo não pode conter ciclos de valor negativo. O algoritmo calcula, para cada par de vértices, o menor de todos os caminhos entre as vértices, considerando o de menor custo [3].

O algoritmo se baseia nos seguintes passos[3]:

- assumindo que os vértices de um grafo orientado G são V = 1, 2, 3, ..., n, considere um subconjunto 1, 2, 3 ..., k;
- Para qualquer par de vértices (i, j) em V, considere todos os caminhos de i a j cujos vértices intermédios pertencem ao subconjunto 1, 2, 3..., k, e p como o mais curto de todos eles;
- O algoritmo explora um relacionamento entre o caminho p e os caminhos mais curtos de i a j com todos os vértices intermédios em 1, 2, 3 ..., k-1;
- O relacionamento depende de k ser ou não um vértice intermédio do caminho p.

Algoritmo de *Floyd* é um problema clássico da computação de alto desempenho devido à sua complexidade. Quando paralelizado, se obtém um ganho considerado de *speedup*. Dessa forma, sua utilidade pode ser aplicada nos seguintes problemas:

- identifica caminhos mais curtos em grafos orientados (algoritmo de Floyd).
- encontra proximidade transitiva de grafos orientados (algoritmo de Warshall).
- encontrar uma expressão regular denotando a linguagem regular aceita por um autômato finito (algoritmo de Kleene);
- inversões de matrizes de números reais (algoritmo de Gauss-Jordan);
- roteamento otimizado.
- computação rápida em *Pathfinder networks*

III. Metodologia e Implementação

Para implementar o algoritmo de *Floyd* utilizou a linguagem C, com as bibliotecas MPI e OMP no padrão mestre-escravo. Os testes iniciais foram realizados

com uma matriz de duas dimensões de 3072 posições. O algoritmo de *Floyd* com mesma dimensão pode ser eficientemente implementado em ambientes distribuídos utilizando MPI. No entanto, nem sempre é possível isto pois depende da dimensão das matrizes a serem calculadas e do número de processos disponíveis. Para matrizes de dimensão NxN e P processos disponíveis no sistema, deve existir um inteiro Q tal que $P = Q*QeNmodQ = 0$. Ou seja, utilizando a divisão dos problemas entre os nodos (MPI), deve-se obter uma divisão exata destas matrizes.

A decomposição dos trabalhos utiliza a distribuição os nodos com a função *broadcast* do MPI. Com as cargas de trabalho rodando em memória compartilhada, o paralelismo é explorado com o OMP utilizando as diretivas de *pragma parallel for*.

Comumente em um padrão de paralelização mestre-escravo, temos que destinar um processo para coordenar os trabalhos enviados aos escravos. Contudo, para tornar o modelo mais eficiente, o código foi desenvolvido para que o mestre também execute uma parte do trabalho, até receber a resposta dos escravos. Desta forma, os testes foram conduzidos através de 2, 4, 6, 8, 16, e 32 processos, utilizando 4 nodos. Ampliando o paralelismo com o OMP, cada processo em cada nodo expande em 2 *threads* as execuções das regiões paralelas. Dessa forma, a distribuição das cargas de trabalho é feita de maneira que cada processo receba uma parte NxN da matriz.

O ambiente utilizado para os testes foi o Cluster Atlântica, localizado no Laboratório de Alto Desempenho (LAD). Cada servidor possuí processadores Xeon Quad-Core E5520 2.27 GHz Hyper Threading (HT) com 16 *cores* incluindo HT e 16 GB de memória RAM.

IV. Resultados dos testes

Algoritmos com complexidades quadráticas ou cubicas, tornam-se muito eficientes quando é possível realizar a paralelização do problema. Pode-se citar outros algoritmos como *Bubble Sort*, que ao escolher o padrão de paralelismo adequado apresenta ganhos exponenciais.

Sabendo disso, os resultados de *speedup* apresentados nessa seção têm ganhos consideráveis. Porém, o método de implantação escolhido neste trabalho deve ser observado alguns aspectos particulares de complexidade, que são o envio e retorno das mensagens e a sincronização dos resultados após o processamento de cada escravo.

A Figura 1 apresenta os resultados de *speedup* e eficiência dos testes executados. O algoritmo é eficiente na utilização de espaço em memória. Contudo, como se trata de uma aplicação *cpu-bound* concentra a maior parte do processamento na comparação dos valores de cada aresta. Desta forma, quando utilizado o recurso de *Hyper Threading* do processador o *speedup* é extremamente afetado devido a concorrência dos *cores* virtuais com os *cores* físicos.

Com o intuito de analisar a relação entre o aumento de processos e o tempo de execução [1], foram coletados os tempos antes do envio das mensagens e calculado a média destes. Conforme o modelo implantado, a utilização de mais processos gera um tempo relativamente maior, como pode ser observado na Tabela I. Isto se dá devido ao fato que cada processo recebe uma parte da matriz para ser

[1]Neste teste foi considerado a execução em uma *thread* somente.

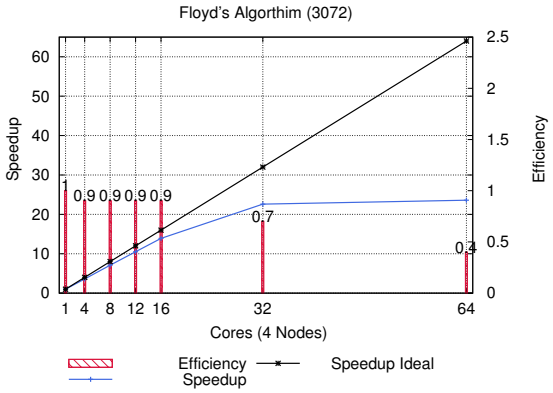

Figura 1: Resultados de *speedup* e eficiência

processada, contudo, por se tratar de uma mensagem *broadcast*, a carga do mestre em controlar essas mensagens acaba afetando o tempo de espera de cada processo. Outro aspecto que deve ser observado é o tempo existente quando se utiliza um único processo, que mesmo executando em um único nodo, o MPI realiza chamadas para interface de rede, porém, não sendo penalizado pelo *delay* do tráfego do pacote.

Tabela I: Relação do aumento de processos com tempo das trocas de mensagens

Nodes	Processes	Avg. Time
4	64 – 4	3.24/0.49
3	48 – 3	2.92/0.45
2	32 – 2	3.18/0.42
1	# – 1	–/0.06

V. Dificuldades Encontradas

Inicialmente, o principal aspecto que dificultou o desenvolvimento foi o entendimento do algoritmo. Mapear e gerenciar o controle dos valores das distâncias foi um dos fatores que impactou na evolução da solução. A decomposição do problema com MPI também causou alguns contratempos. Com o objetivo de utilizar a função *broadcast* do MPI para otimizar o desempenho do código, eram constante os problemas de *deadlocks*, pois se trata de uma função um tanto complexa, pois de maneira análoga nesta função temos as funções *MPI_Send* e *MPI_Rcv*.

VI. Considerações Finais

Este trabalho apresentou resultados de desempenho da implementação híbrida do algoritmo de *Floyd*. O algoritmo apresentou *speedup* e eficiência consideraveis para um algoritmo paralelo, porém penalizado somente pelo recurso HT do processador. Através deste, foi possível analisar que este tipo abordagem híbrida de programação paralela favorece na execução de algoritmos computacionais, pois MPI proporciona a distribuição entre nodos das cargas de trabalhos mais pesadas, e com OMP, o processamento é realizado em memória compartilhada utilizando *threads*. Na literatura existem algumas outras abordagens para paralelização deste algoritmo, por exemplo a *pipeline 2-D block mapping*, que busca amenizar o tempo de sincronização dos resultados.

Referências

[1] Wikipedia, "Floyd-Warshall Algorithm," December 2015. [Online]. Available: $https://en.wikipedia.org/wiki/Floyd\%E2\%80\%93Warshall_algorithm$

[2] T. H. Cormen, *Introduction to Algorithms*. MIT press, 2009.

[3] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *Introduction to Parallel Computing*. Pearson, 2003.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
#include <mpi.h>
#include <math.h>

#define SIZE 3072
#define nThreads 2
//#define SIZE 100
//#define nThreads 4
#define min(a,b) ((a)<(b)?(a):(b))

//Sequencial
int** floyd(int N, int **A) {
    for (int k = 0; k < N; k++) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                A[i][j] = min(A[i][j], A[i][k] + A[k][j]);
            }
        }
    }
    return A;
}

//Inicializa as matrizes
void initMatrix(int firstRow, int numRows, int firstColumn, int numColumn, int ** A)
{
    for (int i = 0, ii = firstRow; i < numRows; i++, ii++)
    {
        for (int j = 0, jj = firstColumn; j < numColumn; j++, jj++)
        {
            A[i][j] = 100;
            if (((jj + 1) % (ii + 1)) == 0)
                A[i][j] = 1;
            if (((ii + 1) % (jj + 1)) == 0)
                A[i][j] = 1;
            if (ii == jj)
                A[i][j] = 0;
        }
    }
}

int** floyd2D(int numRows, int ** block, int * row, int myRank, double * commTime)
{
    int n = SIZE;
    double start_time = 0.0;
    for (int k = 0; k < n; k++)
    {
        #pragma omp single
        {
            if (myRank == 0)
                start_time = MPI_Wtime();
            int p = k / numRows;
            if (p == myRank)
            {
                int kk = k % numRows;
                for (int i = 0; i < n; i++)
                    row[i] = block[kk][i];
            }
            MPI_Bcast(row, n, MPI_INT, p, MPI_COMM_WORLD);
            *commTime += MPI_Wtime() - start_time;
        }

        #pragma omp for schedule(guided)
        for (int i = 0; i < numRows; i++)
        {
            for (int j = 0; j < n; j++)
            {
                block[i][j] = min(block[i][j], block[i][k] + row[j]);
            }
        }
```

```
    }
    return block;
}

int main(int argc, char** argv)
{

    double commTime;
    int n = SIZE;

    //MPI-----------------------------------
    int myRank, nProc;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nProc);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    //MPI-----------------------------------

    int rowsPerProcesses = n / nProc;
    int firstRow = myRank * rowsPerProcesses;

    //OMP-------------
    omp_set_num_threads(nThreads);
    //OMP-------------

    //Alocação de memória-------------------
    int ** block = (int **) malloc(rowsPerProcesses * sizeof (int*));
    block[0] = (int *) malloc((rowsPerProcesses * n) * sizeof (int));
    for (int i = 1; i < rowsPerProcesses; i++)
        block[i] = block[i - 1] + n;
    int * row = (int *) malloc(n * sizeof (int));
    //Alocação de memória-------------------

    //Inicializa as Matrizes
    initMatrix(firstRow, rowsPerProcesses, 0, n, block);
    //---------------------

    floyd2D(rowsPerProcesses, block, row, myRank, &commTime);

    double totalTime = 0.0;
    double seconds = MPI_Wtime();
    int nIterations = 10;

    #pragma omp parallel shared(block, row, commTime)
    {
        int thread_id = omp_get_thread_num();
        floyd2D(rowsPerProcesses, block, row, myRank, &commTime);
        if (myRank == 0 && thread_id == 0)
        {
            totalTime = MPI_Wtime();
            commTime = 0.0;
        }
        for (int i = 0; i < nIterations; i++)
        {
            floyd2D(rowsPerProcesses, block, row, myRank, &commTime);
        }
        if (myRank == 0 && thread_id == 0)
        {
            totalTime = MPI_Wtime() - seconds;

            printf("\n Matrix: %d \n Average Time: %1.6f \n Average Communication Time: %f\n", n,
totalTime / nIterations; , commTime / nIterations;);
        }
    }
    free(row);
    free(block[0]);
    free(block);
    MPI_Finalize();
    return (EXIT_SUCCESS);

}
```