TRABALHO PRÁTICO 6:
Paralelismo

Bruno Maciel Peres

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

brunomperes@dcc.ufmg.br

Resumo. *Este relatório descreve uma solução proposta para a paralelização de um algoritmo que solucione o casamento estável, ou SMP. Os principais objetivos deste trabalho são: exercitar questões ligadas a paralelização de algoritmos, análise da solução proposta e aprofundamento dos conhecimentos na biblioteca da linguagem C pthreads.*

1. INTRODUÇÃO

Neste trabalho, descrevemos e implementamos um algoritmo paralelo para otimizar o problema de casamento estável entre dois grupos com uma lista de prioridades entre si, ou *Stable Marriage Problem* em inglês. O SMP, como será referido a partir de então é um problema conhecido e foco de estudos em Ciência da Computação.

O SMP consiste em encontrar *matchings* estáveis para todos os elementos de dois grupos que possuem uma ordem de preferência sobre os elementos do outro grupo. Um *matching* ocorre quando dois elementos de grupos distintos estão ligados por uma relação (casamento) e é considerado estável quando não há, dentre as opções de preferência de ao menos um dos casados, outra combinação onde o casado e outro da lista de preferência preferem estar casados entre si ao invés de estar seu parceiro atual.

Este algoritmo possui aplicabilidade para resolução de problemas de interesse entre dois grupos. Um exemplo é alocação de estudantes de medicina formandos que preferem determinados hospitais, listando-os em ordem e hospitais que preferem alguns estudantes de medicina, ordenados por suas notas ou pelas áreas mais deficientes do hospital, por exemplo.

A paralelização de um algoritmo visa reduzir o tempo de execução da sua versão sequencial, distribuindo entre cores, threads ou processadores a carga de processamento a ser realizada. A paralelização cria uma carga extra de processamento ou tempo, devido à necessidade de sincronização entre as threads, isso pode levar à ociosidade de uma thread enquanto a outra não termina seu processamento. Esse problema pode ser agravado pelo desbalanceamento de carga entre essas threads.

A paralelização do algoritmo partiu do algoritmo sequencial proposto por Gale-Shapley[5], o mesmo abordado na realização do Trabalho Prático 0. Mais informações sobre esse algoritmo podem ser obtidas nas referências no final deste documento e na seção **Solução Proposta**.

Dado que o problema já fora tratado em sua versão sequencial em outro Trabalho, este relatório focará na comparação das versões sequencial e paralela, mas principalmente na versão paralela, especialmente no que se refere a tempo de execução. Para avaliar a solução proposta utiliza-se as métricas de Speedup e Eficiência.

2. SOLUÇÃO PROPOSTA

A solução sequencial proposta para o SMP é utilizar o algoritmo de Gale-Shapley operando sobre o Tipo Abstrato de Dado lista duplamente encadeada.

O algoritmo se resume em tentar casar todos os homens seguindo suas ordens de preferências. Caso haja algum conflito (um homem deseja casar-se com uma mulher já casada), a lista de preferências dessa mulher é acessada e dentre os dois homens conflitantes, aquele que for de maior interesse para a mulher será casado com essa mulher. O estado de casado do homem rejeitado passa a ser NÃO casado. Na próxima iteração o homem rejeitado tentará ser casado novamente. Quando não houver mais homens solteiros, uma solução estável foi obtida.

A versão paralela dessa solução utiliza o paradigma de programação paralela de Mestre/Escravo, onde uma thread principal atribui tarefas à threads escravas. Essas últimas, à medida que terminam a execução, entregam o resultado à thread mestre.

A thread mestre foi modelada para atribuir a tarefa de casar um homem solteiro à uma thread escrava. A thread escrava deve, por sua vez, percorrer a lista de preferências desse homem, tentando casá-lo com mulheres de sua preferência. Caso uma mulher ou um homem que precise ser acessado por uma thread esteja sendo utilizado por outra thread, a primeira deve aguardar através de *busy-waiting*, implementada através da função *mutex_lock()* na biblioteca *pthreads*.

A complexidade de tempo do algoritmo paralelizado é $O((n/p)^3)$, onde p é a quantidade de threads. É importante considerar a Lei de Amdahl e que à medida que a quantidade de threads aumenta, há maior tempo ocioso entre as threads e há mais operações de sincronização, o que pode levar à um aumento no tempo de execução final para programas com um número muito alto de threads.

2.1. Casamento

A função de casamento realiza o casamento de um homem baseado na sua lista de preferências. Em caso de rejeição de uma mulher já casada, essa opção de mulher é removida da lista de preferências para evitar recálculo caso o programa tenha de casar esse homem novamente.

A complexidade assintótica do pior caso dessa função é $O(n^2)$, na situação em que deverá percorrer toda a lista de preferências e ser rejeitado por todas as mulheres, exceto a última. O cálculo da função de desempate é $O(n)$, pois percorre a lista de preferência da mulher.

No melhor caso a mulher está solteira, executando a função em $O(1)$.

2.2. Versão Sequencial

Na versão sequencial da solução, um loop casa todos os homens enquanto houverem homens solteiros. A complexidade do pior caso da versão sequencial é $O(n^3)$, pois deve

1: Casa homem(id i do homem pretendente)

```
novo nó cursor = primeira opção do homem;
while homem i está casado do
    if mulher i na lista de preferência do homem está casada then
        melhor opção = mulher desempata por sua preferência (maridoatual, pretendente);
        if melhor opção == pretendente then
            descasa(maridoatual, mulher);
            casa com o pretendente;
        else
            cursor = próximo na lista do homem;
            remove mulher i da lista do pretendente;
        end
    else
        casa o homem corrente com a mulher na lista;
        quantidade homens solteiros - -;
    end
end
```

executar $a \times n$ vezes a função *casa_homem()*, de complexidade quadrática.

2: men_purpose_algorithm-sequencial()

```
i = 1;
while quantidade de homens solteiros > 0 do
    if i = 1 then i > N
        ;
    end
    if homem i não está casado then
        casa_homem (i);
    end
    i++;
end
```

2.3. Versão Paralela

A abordagem da versão paralela buscou distribuir a função de casar homem para $p - 1$ threads, uma vez que uma das threads executa a função principal. Apesar das threads obterem acesso exclusivo a algumas listas de preferências de tempos em tempos, à medida em que o valor de n aumenta em uma distribuição aleatória, há menor probabilidade de

haver corrida para acesso aos dados (*data race*).

3: men_purpose_algorithm-paralelo()

```
t = 0 índice das threads;
i = 1;
while quantidade de homens solteiros > 0 do
    if i > n then
        i = 1
    end
    if homem i não está casado then
        nova_thread(casa_homem (i));
        t++;
    end
    if t > (número máximo de threads - 1) then
        join em todas as threads;
        t=0;
    end
    i++;
end
join em todas as threads;
libera memória alocada dinamicamente;
```

3. MODELAGEM

3.1. Estruturas de dados

O problema foi modelado utilizando dois vetores de listas duplamente encadeadas. Cada lista armazena a lista de preferências de cada indivíduo e cada vetor representa um grupo de entidades, homens ou mulheres.

A complexidade de espaço para armazenamento das informações de entrada é $O(2 \times n^2)$, onde n é a quantidade de indivíduos em um dos grupos (já que ambos tem a mesma quantidade de elementos). Assintoticamente, a complexidade de espaço final é $O(n^2)$.

Cada thread armazena somente variáveis locais para controle de iterações ou algumas variáveis inerentes à execução da função, logo foram consideradas como overhead.

3.1.1. Estruturas Globais

Para garantir o acesso das threads aos dados necessários para processamento, os dados são armazenados em uma estrutura declarada globalmente. O conteúdo dessa estrutura é descrito abaixo:

4: Estrutura Global

```
Quantidade de homens solteiros;
Número de Threads;
Vetor de listas Homem;
Vetor de listas Mulher;
```

3.1.2. Mutex

Como o acesso às várias posições no vetor pode ser concorrente entre as threads, foi necessário a utilização de *mutex*, que são variáveis de exclusão mútua, inclusas na biblioteca *pthreads* e que fazem o controle de acesso através das funções de *mutex_lock()* e *mutex_unlock()* utilizando o método de *busy-waiting*.

As variáveis mutex utilizadas são declaradas globalmente e são as seguintes:

5: Mutex
Mutex Global: controla o acesso das variáveis Quantidade de homens solteiros e de Número de threads; **Mutex Mulheres:** Um vetor de mutex, onde cada mutex controla o acesso a cada lista de preferência da mulher; **Mutex Homens:** Um vetor de mutex, onde cada mutex controla o acesso a cada lista de preferências do homem;

4. IMPLEMENTAÇÃO

O critério de separação dos arquivos do programa objetivou a modularização do mesmo, para possível utilização posterior, evitar duplicamento de funções, além de melhorar a legibilidade e organização do mesmo.

4.1. Código

4.1.1. Arquivos .c

- **main.c:** Arquivo principal do programa, contém a chamada de execução da função principal para solução do SMP paralelizado.
- **io.c:** Realiza a comunicação do programa com o ambiente, lendo o arquivo de entrada, armazenando as informações lidas na memória e escrevendo os resultados nos arquivos de saída.
- **heuristica.c:** Define o funcionamento da função principal do programa, incluindo a versão paralela e a versão sequencial.
- **lista.c:** Define funções que operam sobre o TAD lista encadeada.
- **timerlinux.c:** Contém funções para mensurar o tempo de execução do programa.

4.1.2. Arquivos .h

Os arquivos de cabeçalho .h definem as estruturas e o cabeçalho das funções dos arquivos .c respectivos.

4.2. Compilação e execução

O programa deve ser compilado através do compilador GCC através de um *makefile* com o comando *make* ou através do seguinte comando em terminal:

```
gcc main.c lista.c heuristica.c io.c timerlinux.c -o tp6
```

Para execução do programa, são requeridos três parâmetros, o nome do arquivo de entrada, o nome do arquivo de saída, esses dois argumentos podem estar em qualquer ordem. Por último a quantidade de threads a serem utilizadas pelo programa, que deve ser sempre o 5º argumento. Caso não haja ao menos esses 3 argumentos, ou a quantidade de threads seja menor que 1, o programa encerra a execução. A análise de execução do programa pode ser habilitada passando $-a$ para a execução do programa, habilitando a escrita das medidas de avaliação do programa num arquivo de texto. O tamanho máximo de nome de arquivo é 255 caracteres.

Um exemplo é dado a seguir.

```
./tp6 -i input.txt -o output.txt <num_threads>
```

Alternativamente, pode-se executar o programa com as entradas padrões (input.txt, output.txt, 4 threads) utilizando o comando *make run*.

4.3. Entrada e saída

4.3.1. Formato da entrada

O arquivo de entrada cujo nome é passado como argumento para o executável, contém uma linha que define a quantidade de instâncias que este arquivo possui, ou seja, quantas entradas diferentes estão contidas nesse arquivo que deverão ser executadas por esse algoritmo. Na segunda linha, há o número n de indivíduos em cada conjunto, que será, por consequência, a quantidade de elementos contidos naquela linha, já que a lista de preferência percorre todos os elementos do outro conjunto.

Um exemplo de arquivo de entrada é dado a seguir:

```
1
4
2 4 1 3
4 1 2 3
2 3 4 1
2 3 4 1
2 3 4 1
3 4 2 1
3 2 1 4
3 2 4 1
```

4.3.2. Formato da saída

A saída do programa, armazenada em um arquivo de saída *output.txt*, contém informações sobre os n casais formados e a qualidade dos casamentos obtidos através do índice de satisfabilidade. Para cada casamento, é impresso as ids dos homens à esquerda e a id de suas respectivas esposas à direita. Após as ids dos conjunges é apresentado o índice de satisfabilidade geral, a satisfabilidade masculina e a satisfabilidade feminina com precisão de 3 casas decimais. Para o caso de várias instâncias, a sequência se repete abaixo da anterior. Um exemplo de saída é dado abaixo:

```
2 4
1 1
```

```
3 2
4 3
2.250
1.750
2.750
```

5. ANÁLISE EXPERIMENTAL

5.1. Resultados

Para realizar os testes utilizou-se um gerador de entradas escrito em Python que segue anexo à este documento. O gerador foi disponibilizado pelo aluno Rafael R. Cacique durante a realização do TP0. Os tamanhos de entrada foram valores discretos entre $[1000, 7000]$, com variação de 1000 entre cada. 7000 foi o maior valor possível dentro da máquina utilizada para realizar os testes.

Os testes foram realizados em uma máquina com processador Intel Core i3 370M 2.40GHz, 4GiB RAM DDR3 1333MHz e sistema operacional Linux Ubuntu 12.04 x86 versão do kernel 3.2.0-26.

Abaixo uma tabela com os resultados obtidos.

n	**Tempo execução (segs.)**
1000	0.015722
2000	0.068384
3000	0.156200
4000	0.281606
5000	0.444235
6000	0.543542
7000	0.856672

Tabela 1. Resultados para o algoritmo sequencial

# Threads	n	**Tempo execução (segs.)**
4	1000	0.109059
4	2000	0.310657
4	3000	0.537188
4	4000	0.942464
4	5000	1.175335
4	6000	1.456600
4	7000	2.500916

Tabela 2. Resultados com número de threads constante

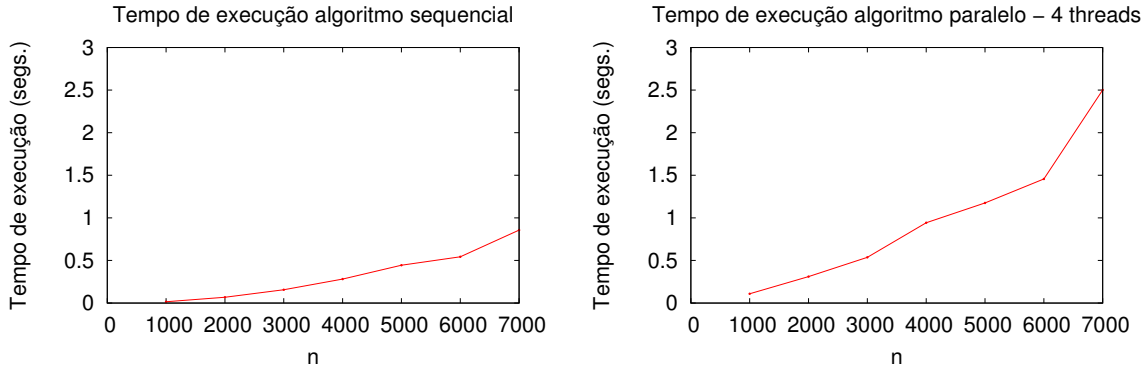
n	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads	64 Threads	128 Threads
1000	0.108846	0.096137	0.079455	0.076632	0.075020	0.083145	0.123748
2000	0.303263	0.259955	0.271714	0.343019	0.192309	0.290873	0.250100
3000	0.542149	0.432848	0.453632	0.472345	0.330278	0.362372	0.384757
4000	0.916267	0.765605	0.780868	0.816397	0.574092	0.649970	0.956197
5000	1.304987	0.932636	0.904193	0.979239	0.708826	0.751331	0.920149
6000	1.464204	1.127131	1.132215	1.120735	0.880291	1.099349	1.321818
7000	2.270563	1.942701	1.876865	1.977839	1.469190	1.843783	2.030872

Tabela 3. Resultados com número de threads quadraticamente crescente

Para o cálculo da eficiência foi-se utilizado os valores para uma entrada de tamanho $n = 7000$.

# Threads	Speedup	Eficiência
2	0,3772949705	0,1886474852
4	0,4409695573	0,1102423893
8	0,4564377299	0,0570547162
16	0,4331353563	0,0270709598
32	0,5830913633	0,0182216051
64	0,464627345	0,0072598023
128	0,4218247137	0,003295505

Tabela 4. Eficiência e Speedup



A partir da análise experimental, nota-se que o tempo de execução é polinomialmente crescente, como esperado da complexidade do algoritmo $O(n^3)$, mantendo-se constante os a quantidade de threads.

De acordo com os valores calculados, o melhor valor de speedup é para 32 threads, mas mesmo assim é mais interessante utilizar o algoritmo sequencial para todos os casos testados.

6. CONCLUSÃO

Neste trabalho foi descrito um algoritmo para encontrar o casamento estável entre dois grupos utilizando programação paralela, utilizando o algoritmo de Gale-Shapley distribuindo o processamento entre várias threads.

Conclui-se que a paralelização do algoritmo através da modelagem proposta não é eficiente, aumentando o tempo de execução, provavelmente pelo tempo ocioso e computação adicional inerentes à paralelização de algoritmos.

Uma decisão a nível de implementação tomada foi a de somente passar a função de execução para a thread se houver processamento significante a ser realizado por ela. Ou seja, se um homem i já estiver casado, não há nada o que ser feito dentro da função. Essa mudança modificou o requisito para a chamada da função Isso reduziu o tempo de ocioso das threads para sincronização do programa, promovendo o balanceamento da carga.

Outra melhoria do algoritmo sequencial do TP0 implementada no TP6 foi a remoção das mulheres já testadas da lista de preferências do homem. Essa operação acontece em $O(1)$ e não requer nenhum *lock* adicional, pois acontece logo que um homem é rejeitado, então o acesso a essa lista já é exclusivo para a thread corrente.

O trabalho atingiu seus principais objetivos: aprofundar os estudos em algoritmos paralelos, analisando seu desempenho em termos de tempo de execução como o speedup e eficiência.

Dentre as adversidades encontradas durante o desenvolvimento desse algoritmo, a maior foi a modelagem do problema em diversas threads, de forma conjunta produzissem um único resultado final, muito provavelmente causado por esse ser um primeiro algoritmo desenvolvido sob essa perspectiva. Outro ponto foi o entendimento da biblioteca pthreads e o tratamento de erros do programa paralelo.

Algumas melhorias que poderiam ser consideradas neste trabalho são:

- Realizar testes de forma a solucionar os erros apresentados ocasionalmente durante a execução do programa. Os erros acontecem de forma aparentemente randômica, mas ao executar o programa novamente o erro não acontece mais.
- Utilizar uma modelagem paralela mais eficiente para esse problema.
- Avaliar o algoritmo mais intensamente de forma a descobrir a causa do tempo de execução semelhante para um número variado de threads. Como por exemplo em um computador com processador com mais threads.

7. REFERÊNCIAS

[1] Cormen, T. (2001). Introduction to algorithms. MIT press.

[2] Ziviani, N. (2007). Projeto de Algoritmos com implementacoes em Pascal e C. Pioneira Thomson Learning.

[3] Blaise Barney, Lawrence Livermore National Laboratory - POSIX Threads Programming - https://computing.llnl.gov/tutorials/pthreads/

[4] Jesper Larsen, University of Copenhagen - A Parallel Approach to the Stable Marriage Problem

[5] Department of Information and Computing Sciences, Faculty of Science, Utrecht University. http://www.cs.uu.nl/docs/vakken/an/handouts/2011-03-03_an-stablemarriage.pdf