

TRABALHO PRÁTICO 1

Passeio do Cavalo Paralelo

Wagner Vinícius Guimarães Lopes

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

wagnerlopes.bh@gmail.com

***Resumo:** Este documento descreve um sistema que encontra um caminho válido para um cavalo de xadrez em um tabuleiro de tamanho $N \times M$. Para que o caminho seja válido o cavalo deve passar em todas as casas do tabuleiro sem repetir nenhuma. O sistema utiliza a biblioteca *pthread* para encontrar o caminho utilizando processamento paralelo. Ao final da execução foram feitas comparações de desempenho entre o algoritmo paralelo e uma outra versão que executa processamento serial com o objetivo de identificar qual seria o speedup ou speeddown da estratégia paralela.*

1 INTRODUÇÃO

Neste documento esta descrito a implementação de um sistema que, dado um tabuleiro de tamanho $N \times M$ e uma posição inicial, tenta encontrar um caminho pelo qual o cavalo do xadrez percorre todas as posições passando por cada uma delas uma única vez. Os principais objetivos do trabalho são implementar e entender o funcionamento de algoritmos que executam processamento paralelo. Este trabalho utiliza como base o algoritmo do Passeio do Cavalo desenvolvido no TP0 e a biblioteca *pthread* para implementar a paralelização.


O núcleo do sistema, que é o algoritmo que busca o caminho do cavalo no tabuleiro utiliza a técnica do *backtracking*. Essa técnica consiste em tentar avançar o máximo possível em uma direção no tabuleiro, se o algoritmo chega em uma posição onde não é possível mais realizar movimentos para completar o caminho, os passos são desfeitos até que se encontre um passo anterior com uma possibilidade nova de caminho. Foram implementadas dois algoritmos utilizando a técnica. O primeiro é um algoritmo utiliza processamento serial. O segundo algoritmo utiliza a biblioteca *pthread* para executar processamento paralelo com o objetivo de tirar maior proveito dos computadores multicore.

2 ANALISE DO PROBLEMA

1. Objetivo:

O objetivo desse trabalho é alterar o algoritmo do Passeio do Cavalo desenvolvido no TP0 para que o processamento seja feito em paralelo, ou seja, deve ser implementado paralelismo de dados e de tarefas. Para essa implementação deve ser utilizada a biblioteca *threads*.

2. Possibilidades de movimentação do cavalo:

		p8		p1		
	p7				p2	
						
	p6				p3	
		p5		p4		

Na análise acima, o cavalo localizado em uma posição “p0” qualquer possui oito possibilidades de movimentação para os pontos destacados em vermelho fazendo as seguintes movimentações na matriz:

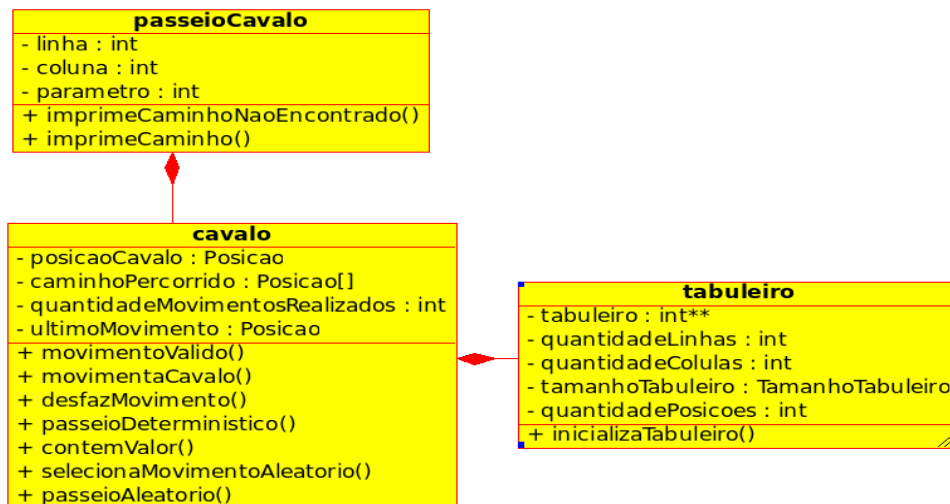
Movimento	deslocamentoLinha	deslocamentoColuna
1	1	2
2	2	1
3	2	-1
4	1	-2
5	-1	-2
6	-2	-1
7	-2	1
8	-1	2

Para realizar a movimentação, criei um vetor de oito posições com os oito deslocamentos possíveis.

3 SOLUÇÃO PROPOSTA

A solução proposta foi a implementação de algoritmos que utilizam o princípio do *backtracking* com recursão e paralelismo. A função que encontra o caminho possui um while responsável por testar todas as possibilidades de movimento partindo de uma determinada posição. Isso possibilita que a pilha de execução do método recursivo armazene somente o sub-caminho que está sendo testado.

Para a implementação dessa solução o sistema foi dividido em três módulos. Um módulo de controle chamado de passeioCavalo, um que representa o cavalo e outro que representa o tabuleiro, como descritos no diagrama de abaixo:



1 Descrição das principais funções:

1.1 cavalo.movimentoValido:

1.1.1 Verifica se o movimento selecionado vai para uma região dentro da área do tabuleiro e, se a posição fizer parte do tabuleiro, verifica se a posição ainda não foi acessada.

1.2 cavalo.movimentaCavalo:

1.2.1 Faz o cavalo avançar um movimento

1.3 cavalo.desfazMovimento:

1.3.1 Faz o cavalo retroceder um movimento

1.4 cavalo.passeio:

1.4.1 Algoritmo de busca pelo caminho. A estratégia se resume em tentar realizar os movimentos em sentido horário. Isso quer dizer que o primeiro movimento testado será o que está na posição 1 hora em relação a posição atual do cavalo, depois 2h, 4h, 5h, etc.

1.5 tabuleiro.inicializaTabuleiro

1.5.1 Aloca espaço para a matriz N x M inicializando todas as posições com zero.

4 Implementação

1. Código

1. Arquivos

1. **passeioCavalo.c**: Módulo de controle. Esse módulo trata as entradas de dados, chama a função que implementa a estratégia selecionada pelo usuário e imprime o resultado.
2. **cavalo.h**: Implementa as funcionalidades do cavalo descritas na seção anterior
3. **tabuleiro.h**: TAD que representa o tabuleiro. Possui um vetor bidimensional que representa o tabuleiro, quantidade de linhas, colunas, quantidade de posições e uma função que faz a alocação do tabuleiro de acordo com o tamanho selecionado pelo usuário.

2. Estratégia de implementação do paralelismo

1. Paralelismo de dados:

1. A primeira forma foi o paralelismo de dados. Essa estratégia foi implementada utilizando duas *threads*. Essa decisão foi tomada com base no meu processador que é um dual core. Sendo assim, a criação de mais *threads* resultaria em um maior overhead causando prejuízo ao desempenho do programa. Sendo assim, dividi o vetor de movimentos em pares e ímpares, resultando em dois vetores contendo quatro possibilidades de movimento cada um. Sendo assim, Adicionei uma outra função chamada “controlaPasseio” no módulo do cavalo. Essa função é responsável por encontrar os movimentos válidos do subvetor de movimentos e iniciar o passeio do cavalo a partir dessa posição. Assim que o passeio é inicializado, o cavalo continua a busca pela solução utilizando um vetor integral de movimentos.

Para que uma thread não interferisse no trabalho da outra, criei uma cópia do tabuleiro para cada um dos subprocessos. Dessa forma, as threads conseguem trabalhar bem sem a necessidade do uso de semáforos, exclusão mútua ou qualquer outra ferramenta de sincronia que pudesse causar overheads. Essa decisão foi tomada baseando-se no fato de que a utilização de memória pelo programa é muito pequena e, por outro lado, o maior desafio seria encontrar uma forma mais rápida para encontrar a solução do problema. Para finalizar as threads utilizei uma variável compartilhada que é inicializada com -1 (menos um). Quando o caminho é encontrado, essa variável é alterada para armazenar o identificador da *thread* que o encontrou. Esse identificador, além de informar que a solução foi encontrada, também define o índice do tabuleiro que será impresso (tabuleiro zero ou um). Dessa forma, a outra *thread* pode saber que a solução foi encontrada e que o trabalho pode ser finalizado.

2. Paralelismo de tarefas:

1. Uma das maiores dificuldades do projeto foi em contrar uma possibilidade de implementação do paralelismo de tarefas.

Inicialmente, tentei implementar *threads* para realizar os movimentos do cavalo, porém, essa estratégia foi frustrada porque a maior parte do código foi identificada como seção crítica e se estendia por 3 funções: cavalo.movimentoValido,

cavalo.desfazMovimento e cavalo.movimentaCavalo. Como essas três funções executam a maior parte do processamento, utilização do mutex nessas seções críticas resultou em uma utilização média de 20% do poder de processamento de cada um dos processadores e inviabilizando totalmente a estratégia.

Sendo assim, o paralelismo de tarefas se resumiu à função que imprime o resultado da execução.

3. Compilação

1. Acesse a pasta “Compila”.
2. Digite “make”.
3. É gerado o executável dentro da pasta “Compila”

4. Execução

1. Acesse a pasta “Compila”
2. Digite:

```
1. ./passeioCavalo -n <linas> -m <colunas> -i <linhaInicial> -j <colunaInicial> -s <arquivoSaida>
```

5. Formado da saída

Solução:

5	8	3	18	15	12	21	24
2	17	6	9	22	19	14	11
7	4	1	16	13	10	23	20

Métricas de performance:

Total de movimentos realizados: 57417

Quantidade de movimentos antes do primeiro retrocesso: 20

Tempo de execucao: 0.030s

5 ANALISE DE COMPLEXIDADE

Como descrito na seção “SOLUÇÃO PROPOSTA”, os algoritmos que analisam o tabuleiro são algoritmos recursivos mas que possuem um while interno que testa todas as oito possibilidades de movimentação partindo de uma determinada posição. Encontraremos a função de complexidade dada pelo pior caso de execução. Considerando que a posição anterior do caminho do cavalo nunca será válida, esse pior caso acontecerá, quando o algoritmo testar todas as possibilidades antes de descobrir que o problema não tem solução para os dados de entrada. Nesse caso, independente se o cavalo está na borda do tabuleiro ou está no centro, o algoritmo implementado irá tentar caminhar por todas as possibilidades de movimento. Nesse caso, se considerarmos sete possibilidades para cada casa, teremos a seguinte função de complexidade:

Função de complexidade = $7^{\text{colunas} \times \text{linhas}}$

Ou seja, temos um algoritmo de complexidade exponencial $O(7^{N \times M})$. Isso dificulta muito a análise de tabuleiros de tamanho maior, demandando muito tempo para que se consiga uma resposta

6 PROBLEMAS DURANTE A IMPLEMENTAÇÃO

A maior dificuldade para a implementação foi definir e implementar estruturas de controle para que as threads não se atrapalhassem durante a busca por solução. Durante esse processo, ocorreram vários problemas de acesso indevido à memória. Esse problema foi solucionado com a ajuda da ferramenta valgrind que foi usada para identificar os pontos onde ocorriam as falhas de nos acessos à memória. Outra grande dificuldade foi encontrar e entender o funcionamento das threads para que o paralelismo pudesse ser implementado.

7 AVALIAÇÃO EXPERIMENTAL

Nessa seção vamos avaliar e comparar a estratégia paralela e a serial de busca pelo caminho no tabuleiro. Para isso, o sistema coleta informações de execução que serão exibidas junto com a resposta do algoritmo de busca. Essas informações coletadas estão listadas abaixo:

Total de movimentos realizados

Quantidade de movimentos antes do primeiro retrocesso

Tempo de execucao

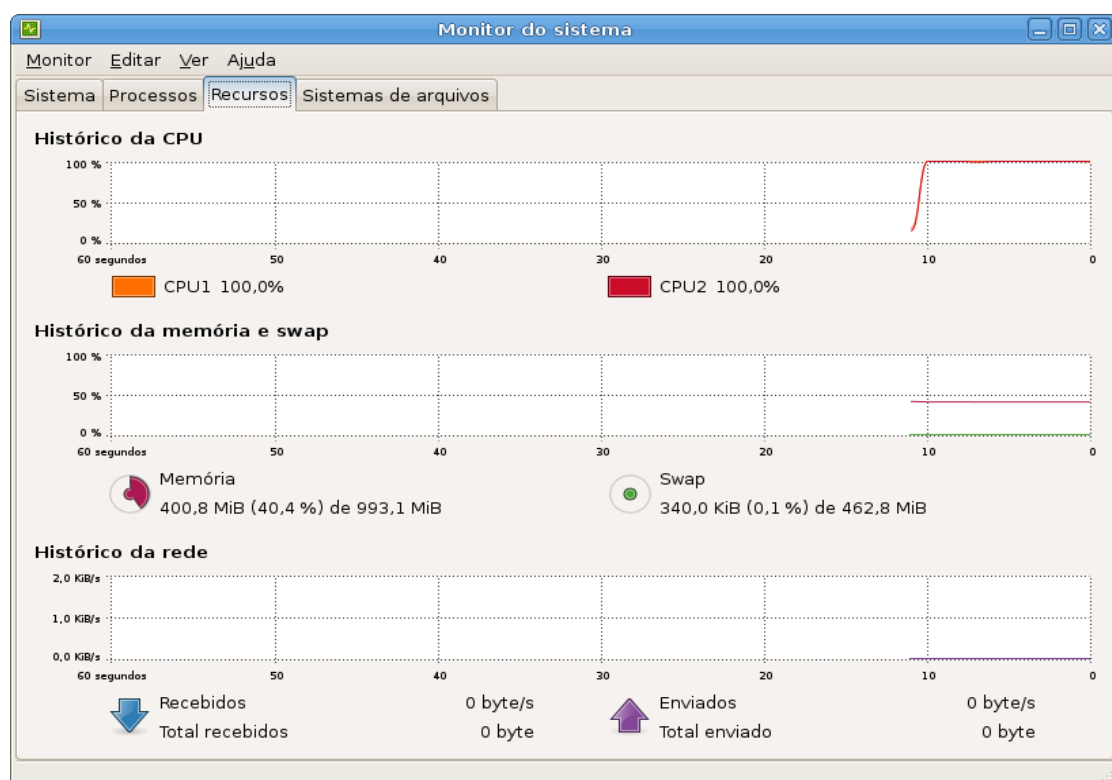
Os testes foram executados em um computador com a seguinte configuração

Processador: Intel Dual-Core 1,86GHZ – 1MB cache

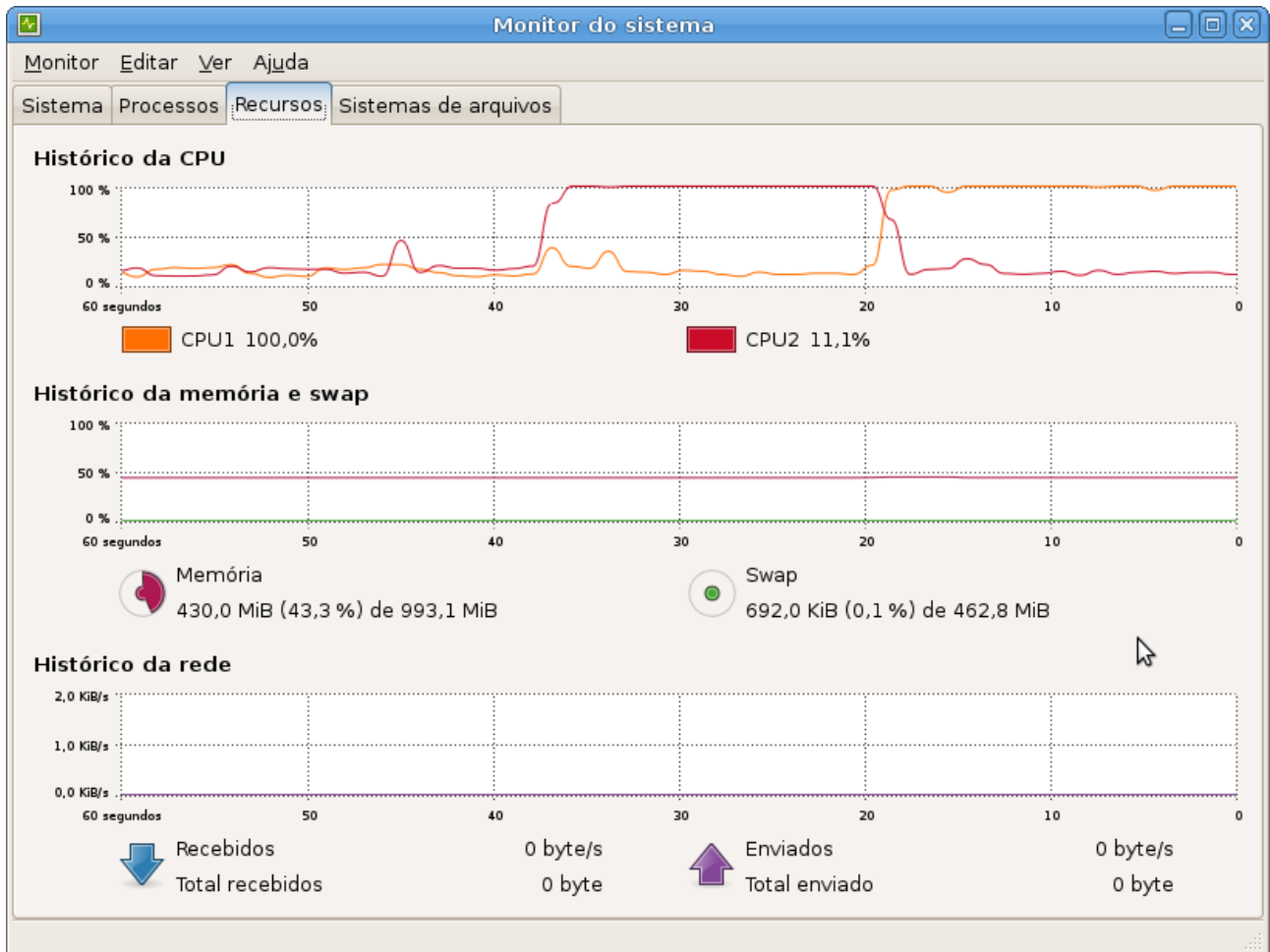
Memória: 1GB

6. Comportamento do sistema ao executar os programas paralelo e serial:

Printscreen com os dois processadores trabalhando em 100% ao mesmo tempo por causa do processamento paralelo.



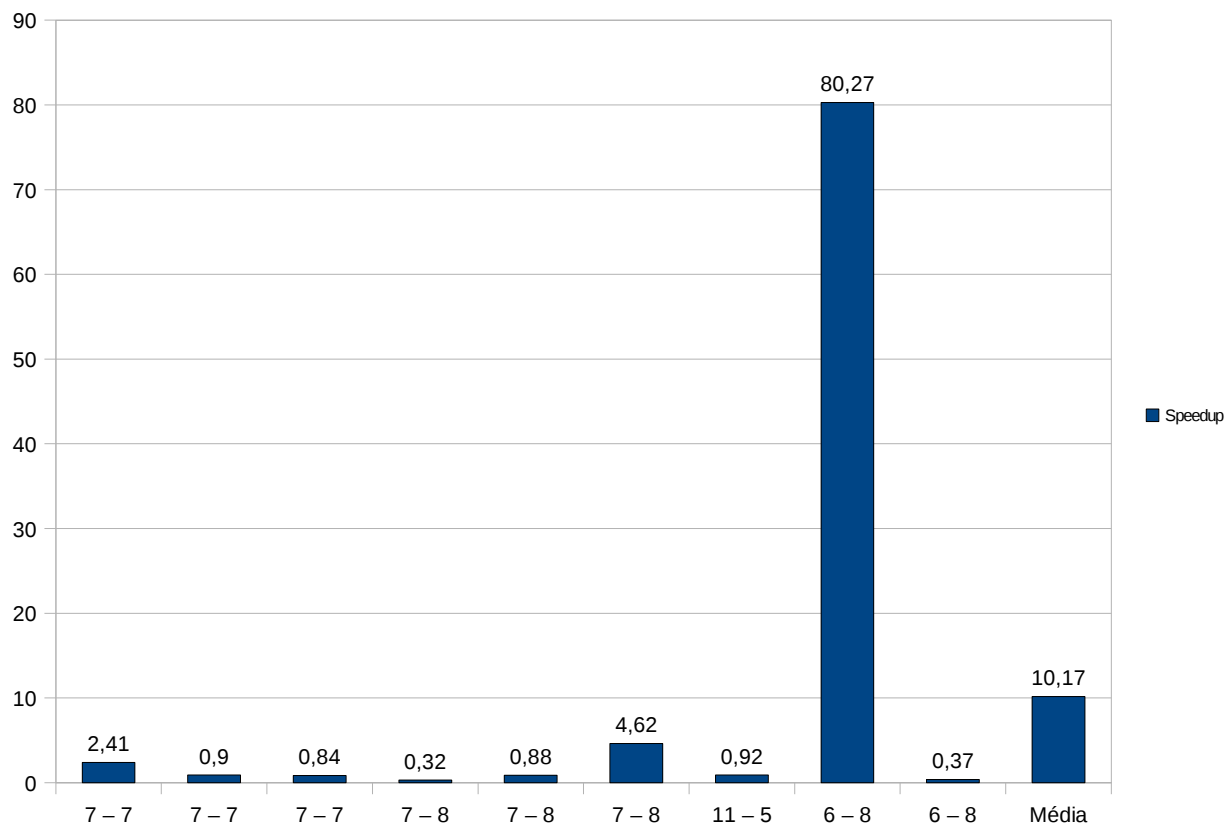
Printscreen com somente um processador trabalhando em 100% por causa do processamento serial:



COMPARAÇÃO ENTRE ESTRATÉGIAS:

Para os testes foram selecionados tamanhos variados de tabuleiro e os resultados estão na tabela e no gráfico abaixo:

Linhas – Colunas	Início	Total de Movimentos Serial	Total Movimentos Paralelo	Tempo Serial	Tempo Paralelo	Número Thread	Speedup
7 – 7	1 – 1	14302308	3689419	1,83	0,76	1	2,41
7 – 7	2 – 2	121397680	137265945	14,85	16,45	0	0,9
7 – 7	3 – 3	3021262	3021263	0,41	0,49	0	0,84
7 – 8	1 – 1	79963325	153492694	9,62	30,07	1	0,32
7 – 8	2 – 2	403676859	403676860	47,66	53,91	0	0,88
7 – 8	3 – 3	5739869495	774111819	698,46	151,26	1	4,62
11 – 5	1 – 1	19398726	19398727	2,46	2,68	0	0,92
6 – 8	1 – 1	346947150	2414250	44,15	0,55	1	80,27
6 – 8	2 – 2	1549379	2616396	0,23	0,62	0	0,37
Média							10,17



Como podemos perceber pelos dados acima, o *overhead* causado pelo paralelismo causou um aumento do tempo de processamento nos caminhos encontrados pela *thread* 0. Essa *thread* percorre exatamente o mesmo caminho do programa serial. Porém, quando o caminho é encontrado pela *thread* 1, que percorre um caminho diferente, percebemos um ganho muito significativo no tempo de processamento, resultando em um *speedup* médio de 10,17 para os casos de teste acima.

Pelos dados acima, é possível perceber que o algoritmo paralelo implementado leva vantagem nos seguintes casos:

- Tabuleiros onde o primeiro movimento não leva a caminho válido:
 - Nesse caso, enquanto o programa serial teria que testar todas as possibilidades a partir do primeiro movimento, o programa paralelo já estaria testando um segundo caminho

totalmente diferente e encontraria a solução muito antes do programa serial.

- Tabuleiros onde é mais fácil encontrar a solução pelo caminho da thread 1 do programa paralelo:
 - Nesse caso, enquanto o programa serial levaria mais tempo procurando em um caminho mais longo, o programa paralelo poderia encontrar uma solução muito mais rapidamente pela segunda thread. Esse comportamento pode ser facilmente percebido na tabela acima, onde a *thread* 1 encontrou caminhos muito mais rápidos que o programa paralelo, sendo que, no penúltimo caso de teste, o *speedup* foi maior que 80.
- Problema que não tem solução para os dados de entrada
 - Nesse caso, o processamento paralelo seria totalmente aproveitado e o programa paralelo, teoricamente, levaria pouco mais da metade do tempo do serial para chegar a essa conclusão. Isso aconteceria porque cada um dos processadores iria testar a metade do tabuleiro no programa paralelo e no serial o tabuleiro inteiro teria que ser testado por um único processador.

8 CONCLUSÃO

A estratégia de paralelização escolhida, deixou, o programa um pouco mais lento que o programa serial em alguns casos porque, na maioria das situações, o caminho já era encontrado tomando o caminho do primeiro movimento realizado, ou seja, o primeiro movimento realizado raramente era desfeito e levava a um caminho válido. Porém, quando o problema caiu em algumas das três situações de vantagens listadas na seção anterior, o algoritmo paralelo se tornou muito mais rápido, resultando em um *speedup* muito significativo.

9 REFERÊNCIAS

Livro: Título: Projeto de algoritmos: Com implementações em Pascal e C
Autor: Ziviane, Nivio.
Publicação: São Paulo - 1993